Algoritmos a fondo

CON IMPLEMENTACIONES EN C y JAVA

INCLUYE DIAGRAMAS DE FLUJO y UML

ING. PABLO AUGUSTO SZNAJDLEDER

Contenidos Interactivos Web



Algoritmos a fondo

Con implementaciones en C y Java

Ing. Pablo Augusto Sznajdleder



Sznajdleder, Pablo
Algoritmos a fondo : con implementaciones en C y Java . - 1a ed. - Buenos Aires : Alfaomega Grupo Editor Argentino, 2012
576 p. ; 24x21 cm.

ISBN 978-987-1609-37-6

1. Informática. I. Título
CDD 005.3

Edición: Damián Fernández
Revisión de estilo: Vanesa García y Juan Micán
Diagramación: Diego Ay
Revisión de armado: Vanesa García

Internet: http://www.alfaomega.com.mx

ISBN 978-987-1609-37-6

Queda hecho el depósito que prevé la ley 11.723

Empresas del grupo:

Argentina: Alfaomega Grupo Editor Argentino, S.A.
Paraguay 1307 P.B. “11”, Buenos Aires, Argentina, C.P. 1057
Tel.: (54-11) 4811-7183 / 0887 - E-mail: ventas@alfaomegaeditor.com.ar

México: Alfaomega Grupo Editor, S.A. de C.V.
Pitágoras 1139, Col. Del Valle, México, D.F., México, C.P. 03100
Tel.: (52-55) 5575-5022 - Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A.
Carrera 15 No. 64 A 29, Bogotá, Colombia
PBX (57-1) 2100122 - Fax: (57-1) 6068648 - E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A.
Doctor La Sierra 1437 - Providencia, Santiago, Chile
Tel.: (56-2) 235-4248 - Fax: (56-2) 235-5786 - E-mail: agechile@alfaomega.cl



Superman, por Octaviano Sznajdleder

A los amores de mi vida: mi esposa Analía y mi hijo Octaviano.
Ellos son el motor que impulsa todo lo que hago.

A la memoria de Naum, quien pasó a vivir en nuestros corazones.
Siempre me preguntaba: —¿Cómo va ese libro, Pablo?

Agradecimientos

A mi mamá Nélida, que no solo pone su casa a mi disposición sino que, además, siempre me prepara el té.

A mi editor y amigo Damián Fernández, que no para de ofrecerme inmejorables oportunidades.

A Graciela Sosisky y Domingo Mandrafina quienes, hace ya varios años, me dieron la oportunidad de incorporarme a la cátedra de Algoritmos.

A Adriana Adamoli por su colaboración y por el aporte de muchos de los ejercicios que se encuentran en la Web.

A Juan Grande quien, GTalk mediante, siempre estuvo presente para darme una mano.

A Marcelo Grillo, Gustavo Baez y a todos los promotores de Alfaomega por la amabilidad, la cordialidad y la excelente estadía que me han hecho pasar durante mi viaje a México.

A los maestros Alberto Templos Carbajal, Sergio Fuenlabrada y Edna Miranda por sus valiosísimos aportes.

Mensaje del Editor

Los conocimientos son esenciales en el desempeño profesional. Sin ellos es imposible lograr las habilidades para competir laboralmente. La universidad o las instituciones de formación para el trabajo ofrecen la oportunidad de adquirir conocimientos que serán aprovechados más adelante en beneficio propio y de la sociedad. El avance de la ciencia y de la técnica hace necesario actualizar continuamente esos conocimientos. Cuando se toma la decisión de embarcarse en una vida profesional, se adquiere un compromiso de por vida: mantenerse al día en los conocimientos del área u oficio que se ha decidido desempeñar.

Alfaomega tiene por misión ofrecerles a estudiantes y profesionales conocimientos actualizados dentro de lineamientos pedagógicos que faciliten su utilización y permitan desarrollar las competencias requeridas por una profesión determinada. Alfaomega espera ser su compañera profesional en este viaje de por vida por el mundo del conocimiento.

Alfaomega hace uso de los medios impresos tradicionales en combinación con las tecnologías de la información y las comunicaciones (IT) para facilitar el aprendizaje. Libros como este tienen su complemento en una página Web, en donde el alumno y su profesor encontrarán materiales adicionales, información actualizada, pruebas (test) de autoevaluación, diapositivas y vínculos con otros sitios Web relacionados.

Esta obra contiene numerosos gráficos, cuadros y otros recursos para despertar el interés del estudiante, y facilitarle la comprensión y apropiación del conocimiento.

Cada capítulo se desarrolla con argumentos presentados en forma sencilla y estructurada claramente hacia los objetivos y metas propuestas. Cada capítulo concluye con diversas actividades pedagógicas para asegurar la asimilación del conocimiento y su extensión y actualización futuras.

Los libros de Alfaomega están diseñados para ser utilizados dentro de los procesos de enseñanza-aprendizaje, y pueden ser usados como textos guía en diversos cursos o como apoyo para reforzar el desarrollo profesional.

Alfaomega espera contribuir así a la formación y el desarrollo de profesionales exitosos para beneficio de la sociedad.

Pablo Augusto Sznajdleder

Es Ingeniero en Sistemas de Información, egresado de la Universidad Tecnológica Nacional (UTN-FRBA) en 1999.

Actualmente, es profesor en la cátedra de “Algoritmos y Estructura de Datos” en la UTN-FRBA, pasando también por la Universidad Nacional de San Martín (UNSAM) y el Instituto de Tecnología ORT Argentina.

Trabajó como instructor Java para Sun Mycrosystems, Oracle e Informix/IBM entre otras empresas líderes.

Desde 1995 trabaja en sistemas, principalmente, en el desarrollo de aplicaciones empresariales distribuidas: primero en C/C++ y luego, en Java/JEE.

En 1996 comenzó a trabajar con Instructor Java para Sun Microsystems y, desde el 2000, se desempeña como consultor en la búsqueda y selección de RRHH capacitados en dicha tecnología, poniendo especial atención en la identificación de jóvenes estudiantes sin experiencia laboral previa, pero con gran potencial profesional.

Tiene las certificaciones internacionales *Sun Certified Java Programmer* (SCJP, 1997) y *Sun Certified Java Developer* (SCJD, 1998) y, además, está certificado como Instructor Oficial Java por Sun Microsystems (1997).

En el 2008 publicó *HolaMundo.pascal*, *Algoritmos y estructura de datos* cuyo contenido cubre por completo los temas que abarca la asignatura de igual nombre en UTN-FRBA. En el 2009 participó como revisor técnico en el libro *Análisis y diseño de algoritmos* (López, Jeder, Vega). En el 2010 publicó su libro sobre desarrollo de aplicaciones Java: *Java a fondo, Estudio del lenguaje y desarrollo de aplicaciones.*



Revisor técnico: Alberto Templos Carbajal

Es Ingeniero en computación de la Facultad de Ingeniería de la UNAM y ejerce, desde 1983, como Profesor de dicha facultad en las Divisiones de Ingeniería Eléctrica en distintas asignaturas de la carrera de Ingeniería en Computación y Educación Continua. Desde 1986, es Profesor a tiempo completo y ha ocupado los siguientes cargos: Coordinador de las carreras de Ingeniero en Computación y, de manera temporal, de Ingeniero Eléctrico Electrónico e Ingeniero en Telecomunicaciones, Jefe del Departamento de Ingeniería en Computación, Secretario Académico de las Divisiones de Ingeniería Eléctrica y Estudios de Postgrado y Coordinador de Postgrado en la Secretaría de Postgrado e Investigación de la Facultad de Ingeniería. También ha sido cofundador de dos empresas de computación y asesor en esa misma área de diversas compañías.

Actualmente, es miembro de diferentes comités, evaluador de planes de estudio del área de computación para diferentes organismos nacionales y es responsable de un proyecto de investigación e innovación tecnológica sobre el Diseño de algoritmos para robots. Su productividad, principalmente, está orientada a la docencia.

Contenido

Modulo 1
Programación estructurada

Modulo 2

Programación orientada a objetos

Modulo 3
Aplicación práctica



Modulo 4
Conceptos avanzados

Información del contenido de la página Web

El material marcado con asterisco (*) solo está disponible para docentes.

Capítulo 1
Introducción a los algoritmos y a la programación de computadoras

- Mapa conceptual
- Autoevaluación
- Videotutorial:
  - Instalación y uso de Eclipse para C
- Presentaciones*

Capítulo 2
Estructuras básicas de control y lógica algorítmica

- Mapa conceptual
- Autoevaluación
- Videotutorial:
  - Uso del debugger para depurar un programa
- Presentaciones*

Capítulo 3
Funciones, modularización y metodología top-down

- Mapa conceptual
- Autoevaluación
- Videotutorial:
  - Mantener archivos de funciones separados del programa principal
- Presentaciones*

Capítulo 4
Tipos de datos alfanuméricos

- Mapa conceptual
- Autoevaluación
- Presentaciones*

Capítulo 5
Punteros a carácter

- Mapa conceptual
- Autoevaluación
- Presentaciones*

Capítulo 6
Punteros, arrays y aritmética de direcciones

- Mapa conceptual
- Autoevaluación
- Videotutorial:
  - Pasar argumentos en línea de comandos con Eclipse
- Presentaciones*

Capítulo 7
Tipos de datos estructurados

- Mapa conceptual
- Autoevaluación
- Videotutoriales:
  - Algoritmo de la burbuja
  - Algoritmo de la búsqueda binaria
- Presentaciones*

Capítulo 8
Operaciones sobre archivos

- Mapa conceptual
- Autoevaluación
- Videotutoriales:
  - Leer y escribir un archivo
  - Leer y escribir un archivo de registros
- Presentaciones*

Capítulo 9
Tipo Abstracto de Dato (TAD)

- Mapa conceptual
- Autoevaluación
- Presentaciones*

Capítulo 10
Análisis de ejercicios integradores

- Mapa conceptual
- Autoevaluación
- Presentaciones*

Capítulo 11
Estructuras de datos dinámicas lineales

- Mapa conceptual
- Autoevaluación
- Presentaciones*

Capítulo 12
Encapsulamiento a través de clases y objetos

- Mapa conceptual
- Autoevaluación
- Presentaciones*

Registro en la Web de apoyo

Para tener acceso al material de la página Web de apoyo del libro:

1. Ir a la página http://virtual.alfaomega.com.mx
2. Registrarse como usuario del sitio y propietario del libro.
3. Ingresar al apartado de inscripción de libros y registrar la siguiente clave de acceso

4. Para navegar en la plataforma virtual de recursos del libro, usar los nombres de Usuario y Contraseña definidos en el punto número dos. El acceso a estos recursos es limitado. Si quiere un número extra de accesos, escriba a webmaster@alfaomega.com.mx

Estimado profesor: Si desea acceder a los contenidos exclusivos para docentes, por favor contacte al representante de la editorial que lo suele visitar o escribanos a:

webmaster@alfaomega.com.mx

	Videotutoriales sobre temas complementarios de la obra.
	Conceptos para recordar: bajo este icono se encuentran definiciones importantes que refuerzan lo explicado en la página.
	Comentarios o información extra: este ícono ayuda a comprender mejor o ampliar el texto principal
Contenidos Interactivos Web	Contenidos interactivos: indica la presencia de contenidos extra en la Web.

Prólogo

Cuando llegó a nuestras manos el libro de *Algoritmos a fondo* escrito por el Ing. Pablo Augusto Sznajdleder, esperábamos un documento con explicaciones complejas como la mayoría de las obras que abordan el tema pero, por el contrario, conforme leíamos el manuscrito nos encontramos con un texto ameno que lleva al lector en un proceso **paulatino, teórico-metodológico y persistente** en la construcción del conocimiento.

Una construcción del conocimiento **acompasada**, ya que para el autor un tema es verdadero si, y solo si, lleva un proceso de reflexión, que permite evidenciar que el conocimiento es indudable, provocando en el lector pocos sobresaltos y dudas.

El autor conduce con orden sus razonamientos, empezando por los conceptos más simples y fáciles de comprender para ascender poco a poco, de forma gradual, hasta llegar al conocimiento de lo más complejo, e incluso suponiendo un orden entre los que no se preceden naturalmente, provocando en los estudiantes el **rigor teórico-metodológico** que se espera en los especialistas de las áreas de informática y computación.

El texto es **persistente** ya que el autor cuida el hacer recuentos integrales del conocimiento y revisiones amplias de lo aprendido, que le permiten al lector estar seguro de que no se omitió algún aspecto del tema tratado, lo cual evita aburrir al lector con repeticiones innecesarias.

Además, resalta el uso de los diagramas Chapin, que son una versión modificada de los diagramas Nassi-Shneiderman, que como técnica de representación del comportamiento esperado de un algoritmo, permite a los estudiantes establecer un modelo estructural libre de “brincos” o “bifurcaciones” que aumentan, de forma artificial, la complejidad original del algoritmo.

Facilita el proceso de aprendizaje y migración de un lenguaje de programación a otros, ya que presenta los ejemplos y algoritmos en dos lenguajes de programación C y Java, permitiendo al estudiante una transición paulatina de la programación estructurada a la orientación a objetos, apoyándose en este proceso con el uso de los diagramas de UML.

Siendo un tema primordial para la resolución de algoritmos el uso de estructuras de datos, el autor dedica varios capítulos a la discusión de estas, iniciando con las estructuras de datos más simples, pasando por operaciones sobre archivos y terminando con una amplia explicación del uso de las estructuras de datos dinámicos lineales en Java.

El autor, adicionalmente, ofrece videos como complemento a la exposición de temas complejos, con el objetivo de disminuir la dificultad en su comprensión, los cuales pueden ser vistos en Internet. Este recurso permite el desarrollo del aprendizaje autodidacta del estudiante a su propio ritmo y, para el docente, puede ser un recurso didáctico que facilite su labor.

Por su organización y contenido, este libro puede ser utilizado como libro de texto siendo una guía en el proceso de enseñanza-aprendizaje o también, como libro de consulta para una o varias asignaturas ya que los contenidos pueden ser abordados de manera independiente.

El libro puede emplearse en asignaturas tales como Fundamentos de Programación, Estructuras de Datos y claro está, en la asignatura de Algoritmos Computacionales, esto dependerá de los contenidos del programa de estudio correspondiente.

Finalmente, a nuestro parecer, el libro despertará la curiosidad de los estudiantes y los animará a continuar por el camino de la construcción de sistemas de cómputo, por lo cual es posible que este libro sea su primer paso para ingresar al fascinante mundo de las Ciencias Computacionales.

Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
PROFESORES-INVESTIGADORES
INSTITUTO POLITÉCNICO NACIONAL
UNIDAD PROFESIONAL INTERDISCIPLINARIA DE INGENIERÍA
Y CIENCIAS SOCIALES Y ADMINISTRATIVAS

Cd. de México, mayo de 2012

Introducción

Algoritmos a fondo es un libro de nivel universitario pensado para cubrir las necesidades de los alumnos que cursan las materias de Algoritmos I, II y III, teniendo en cuenta el aprendizaje basado en competencias.

La obra comienza desde “cero” explicando los conceptos de “algoritmo”, “estructuras de control”, “variables”, “lenguajes de programación”, etcétera; y llega hasta el análisis e implementación de algoritmos complejos como ser los diferentes algoritmos de ordenamiento (*quicksort*, *heapsort*, etc.), recorridos sobre árboles, grafos y recursión.

Se estudian estructuras de datos estáticas y dinámicas, lineales y no lineales. Comenzando con programación estructurada e implementaciones en lenguaje C, hasta llegar a la programación orientada a objetos con implementaciones en lenguaje Java, previo paso por C++.

El libro se desarrolla como un “curso de programación” donde se guía al alumno en un proceso de aprendizaje durante el cual podrá adquirir la lógica necesaria para diseñar e implementar algoritmos.

Cada capítulo introduce un mayor nivel de dificultad, ya sea incorporando nuevos conceptos y recursos, o bien profundizando en técnicas de programación más complejas.

El autor remarca la idea de “curso de programación” ya que esta es totalmente opuesta al concepto de “libro tradicional” que, generalmente, presenta una estructura mucho más rígida, tipo “diccionario” o “enciclopedia”.

Cualquier alumno universitario debería poder leer el libro y aprender a programar por sí solo sin experimentar ningún tipo de dificultad ya que este es uno de los principales objetivos del libro.

La obra se complementa con una serie tutoriales grabados en video en los que el autor explica temas que dada su naturaleza resultarían extremadamente tediosos de leer.

Algoritmos a fondo se compone de cuatro módulos que agrupan los diferentes capítulos, según el siguiente criterio:

El Módulo 1 “Programación estructurada”, comienza desde “cero” y llega hasta el estudio de “estructuras dinámicas lineales”, pasando por “administración de archivos”, *arrays*, TADs, etcétera.

La implementación de los algoritmos y conceptos que aquí se estudian se basa en los diagramas de Chapín; la estructura de estos diagramas es mucho más rígida que la del diagrama de flujo tradicional y ayuda al alumno a pensar en algoritmos compuestos por bloques de “única entrada” y “única salida”. Este razonamiento constituye una de las premisas fundamentales de la programación estructurada.

Todos los algoritmos y programas que se exponen en el libro cuentan con su correspondiente codificación en lenguaje C, documentada y explicada.

Dentro del módulo 1, se incluyen capítulos que explican el lenguaje programación C que, como sabemos, es suficientemente complejo por sí mismo. Principalmente, el manejo cadenas de caracteres y los conceptos de “dirección de” y “contenido de” para poder pasarle argumentos por referencia a las funciones.

Demás está decir que el concepto de modularización es fundamental. Tanto es así que se estudia a partir del Capítulo 3 y se aplica de ahí en adelante.

En el Módulo 2 “Programación orientada a objetos”, se explican los conceptos de programación orientada a objetos, comenzando por la idea de “encapsulamiento”. Es decir: diseñar clases cuyos métodos encapsulen algoritmos complejos de forma tal un programador con menos conocimientos o menos experiencia los pueda utilizar sin tener que preocuparse por comprender su implementación.

Durante este módulo se realiza la transición entre los lenguajes C y Java; para amortiguar este proceso se estudia también algo de código C++.

En C++ se explican los conceptos de “clase y objeto”, encapsulamiento y generalizaciones mediante el uso de *templates*.

Los temas fuertes de la programación orientada a objetos como ser polimorfismo, *interfaces*, factorías, colecciones y clases genéricas se tratan directamente en Java.

El Módulo 3 “Aplicación práctica” es, en sí mismo, un ejercicio integrador cuyo desarrollo requerirá aplicar gran parte de los conocimientos adquiridos durante los dos módulos anteriores. Estamos hablando de un programa compresor/descompresor de archivos basado en el “algoritmo de Huffman”. Aquí se obtendrá suficiente evidencia de las competencias adquiridas hasta el momento.

Esta aplicación constituye un excelente ejercicio que inducirá al alumno a aplicar los principales conceptos estudiados desde el comienzo del libro: *arrays*, archivos y listas.

El algoritmo utiliza un árbol binario para generar los códigos Huffman que reemplazarán a los *bytes* de la fuente de información original. El hecho de que el tema “árbol binario” aún no haya sido estudiado representa una excelente oportunidad para aplicar los conceptos de abstracción y encapsulamiento expuestos en los capítulos del Módulo 2 sobre programación orientada a objetos.

El Módulo 3 se complementa con un *setup* que incluye clases utilitarias con las que el alumno podrá recorrer un árbol binario, leer y escribir “bit por bit” en un archivo, etcétera.

En el Módulo 4 “Conceptos avanzados”, se estudian algoritmos y estructuras de datos que revisten mayor nivel de complejidad.

Comenzando por el tema de recursión, se comparan las implementaciones recursivas e iterativas de diferentes funciones. Por ejemplo, el caso típico de la función *factorial* y el caso extremo de la función de *Fibonacci* cuya versión recursiva es incapaz de resolver, en un tiempo razonable, los términos de la serie superiores a 50.

Se analizan estructuras de datos no lineales: árboles y grafos; siempre aplicándolas a la solución de casos cotidianos para captar el interés del lector. Por ejemplo, una estructura de datos y un algoritmo que permitan implementar un *autosuggest* como el que utiliza Google en su buscador.

Durante los diferentes capítulos de este módulo, se explican algoritmos complejos como ser los diferentes métodos de ordenamiento, implementaciones alternativas mediante “programación dinámica” y los algoritmos tradicionales que operan sobre grafos: Dijkstra, Prim y Kruskal.

El docente debe saber que, en cada capítulo, se mencionan las competencias específicas a desarrollar y que en la página Web del libro dispone de una guía detallada de las competencias que se desarrollan a lo largo del libro y las evidencias que se pueden recolectar.

1

Introducción a los algoritmos y a la programación de computadoras

Contenido

Objetivos del capítulo

- Entender los conceptos básicos sobre algoritmos y programación.
- Identificar los principales recursos lógicos y físicos que utilizan los programas.
- Estudiar el teorema de la programación estructurada y las estructuras de control que este teorema describe.
- Conocer las herramientas imprescindibles de programación: lenguaje de programación, compilador, entorno de desarrollo (IDE), etcétera.
- Desarrollar, compilar y ejecutar nuestro primer programa de computación.

Competencias específicas

- Dominar los conceptos básicos de la programación.
- Analizar problemas y representar su solución mediante algoritmos.
- Conocer las características principales del lenguaje C.
- Codificar algoritmos en el lenguaje de programación C.
- Compilar y ejecutar programas.

1.1 Introducción

Este capítulo introduce al lector en los conceptos iniciales sobre algoritmos y programación. Aquellos lectores que nunca han programado encontrarán aquí los conocimientos básicos para hacerlo.

Los contenidos son netamente introductorios ya que pretenden brindarle al lector las pautas, las expresiones y la jerga que necesitará conocer para poder leer y comprender los capítulos sucesivos.

Por lo anterior, la profundidad con la que se tratan los temas aquí expuestos es de un nivel inicial, ya que cada uno de estos temas será analizado en detalle llegado el momento, en los capítulos que así lo requieran.

1.2 Concepto de algoritmo

1.2.1 Definición de algoritmo y problema

Llamamos "algoritmo" al conjunto finito y ordenado de acciones con las que podemos resolver un determinado problema.

Llamamos "problema" a una situación que se nos presenta y que, mediante la aplicación de un algoritmo, pretendemos resolver.

Los algoritmos están presentes en nuestra vida cotidiana y, aún sin saberlo, aplicamos algoritmos cada vez que se nos presenta un problema sin importar cuál sea su grado de complejidad.

Para ejemplificar esto imaginemos que estamos dando un paseo por la ciudad y que, al llegar a una esquina, tenemos que cruzar la calle. Intuitivamente, aplicaremos el siguiente algoritmo:

- Esperar a que la luz del semáforo peatonal esté en verde (`esperarSemaforo`);
- Cruzar la calle (`cruzarCalle`);

Este algoritmo está compuesto por las acciones: `esperarSemaforo` y `cruzarCalle`, en ese orden, y es extremadamente simple gracias a que cada una de estas engloba a otro conjunto de acciones más puntuales y específicas. Por ejemplo:

La acción `esperarSemaforo` implica:

- Observar la luz del semáforo (`observarLuz`);
- **Si** la luz está en "rojo" o en "verde intermitente" **entonces**
  - esperar un momento (`esperar`);
  - **ir a**: `observarLuz`;

Por otro lado, la acción `cruzarCalle` implica:

- Bajar la calzada (`bajarCalzada`);
- Avanzar un paso (`avanzarPaso`);
- **Si** la distancia hacia la otra calzada es mayor que la distancia que podemos avanzar dando un nuevo paso **entonces**
  - **ir a**: `avanzarPaso`;
- Subir la calzada de la vereda de enfrente (`subirCalzada`);

Como vemos, cada acción puede descomponerse en acciones más puntuales y específicas. Por lo tanto, cada una de las acciones del algoritmo se resuelve con su propio algoritmo o secuencia finita y ordenada de acciones.

Cuando una acción se descompone en acciones más puntuales y específicas decimos que es un “módulo”.

Podemos seguir profundizando cada acción tanto como queramos. Por ejemplo, analicemos la acción (o el módulo) `bajarCalzada`: esta acción se resuelve:

- levantando un pie,
- adelantándolo,
- inclinando el cuerpo hacia adelante,
- y apoyando el pie en la calle,
- luego levantando el otro pie,
- adelantándolo y
- apoyándolo en la calle.

Claro que “levantar un pie” implica tensionar un conjunto de músculos, etcétera.



Módulo
Cuando una acción se descompone en acciones más puntuales y específicas la llamamos "módulo".

1.2.2 Análisis del enunciado de un problema

Resulta evidente que, si vamos a diseñar un algoritmo para resolver un determinado problema, tenemos que tener totalmente estudiado y analizado el contexto de dicho problema. Esto implica:

- Comprender el alcance.
- Identificar los datos de entrada.
- Identificar los datos de salida o resultados.

El análisis anterior es fundamental para poder diseñar una estrategia de solución que será la piedra fundamental para el desarrollo del algoritmo.

1.2.2.1 Análisis del problema

Repasemos el contexto del problema de “cruzar la calle” que formulamos más arriba:

Caminando por la ciudad llegamos a una esquina y pretendemos cruzar la calle. Para poder cruzar con la mayor seguridad posible, tenemos que esperar que el semáforo habilite el paso peatonal. Hasta que esto ocurra nos encontraremos detenidos al lado del semáforo y una vez que este lo permita avanzaremos, paso a paso, hasta llegar a la vereda de enfrente. Se considera que el semáforo habilita el paso peatonal cuando emite una luz “verde fija”. En cambio si el semáforo emite una luz “roja” o “verde intermitente” se recomienda esperar.

1.2.2.2 Datos de entrada

En este contexto podemos identificar los siguientes datos de entrada:

- Nuestra posición inicial - posición en donde nos detuvimos a esperar a que el semáforo habilite el paso peatonal. La llamaremos `posInicial`.
- Luz del semáforo - el color de la luz que el semáforo está emitiendo. Lo llamaremos `luz`.
- Distancia de avance de paso - que distancia avanzamos cada vez que damos un nuevo paso. La llamaremos `distPaso`.
- Posición de la vereda de enfrente. La llamaremos `posFinal` ya que es allí hacia donde nos dirigimos.

1.2.2.3 Datos de salida

Si bien en este problema no se identifican datos de salida resulta evidente que luego de ejecutar la secuencia de acciones del algoritmo nuestra posición actual deberá ser la misma que la posición de la vereda de enfrente. Es decir, si llamamos `p` a nuestra posición actual resultará que su valor al inicio del algoritmo será `posInicial` y al final del mismo deberá ser `posFinal`.

1.2.3 Memoria y operaciones aritméticas y lógicas

En la vida real, cuando observamos la luz del semáforo almacenamos este dato en algún lugar de nuestra memoria para poder evaluar si corresponde cruzar la calle o no.

Generalmente, los datos de entrada ingresan al algoritmo a través de una acción que llamamos “lectura” o “ingreso de datos” y permanecen en la memoria el tiempo durante el cual el algoritmo se está ejecutando.

Por otro lado, el hecho de evaluar si corresponde o no cruzar la calle implica realizar una operación lógica. Esto es: determinar si la proposición “el semáforo emite luz roja o verde intermitente” resulta ser verdadera o falsa.

También realizaremos una operación lógica cuando estemos cruzando la calle y tengamos que determinar si corresponde dar un nuevo paso o no. Recordemos que llamamos `p` a nuestra posición actual, `posFinal` a la posición de la vereda de enfrente y `distPaso` a la distancia que avanzamos dando un nuevo paso. Por lo tanto, corresponde dar otro paso si se verifica que `posFinal > p+distPaso`. Aquí además de la operación lógica estamos realizando una operación aritmética: la suma `p+distPaso`.

En resumen, estos son los recursos que tenemos disponibles para diseñar y desarrollar algoritmos:

- La memoria.
- La posibilidad de realizar operaciones aritméticas.
- La posibilidad de realizar operaciones lógicas.

1.2.4 Teorema de la programación estructurada

Este teorema establece que todo algoritmo puede resolverse mediante el uso de tres estructuras básicas llamadas “estructuras de control”:

- La “estructura secuencial” o “acción simple”.
- La “estructura de decisión” o “acción condicional”.
- La “estructura iterativa” o “acción de repetición”.

Dos de estas estructuras las hemos utilizado en nuestro ejemplo de cruzar la calle. La estructura secuencial es la más sencilla y simplemente implica ejecutar una acción, luego otra y así sucesivamente.

La estructura de decisión la utilizamos para evaluar si correspondía dar un nuevo paso en función de nuestra ubicación actual y la ubicación de la vereda de enfrente.

Sin embargo, para resolver el problema hemos utilizado una estructura tabú: la estructura “**ir a**” o, en inglés, “*go to*” o “*goto*”. Esta estructura quedó descartada luego de que el teorema de la programación estructurada demostrara que con una adecuada combinación de las tres estructuras de control antes mencionadas es posible resolver cualquier algoritmo sin tener que recurrir al “*goto*” (o estructura “ir a”).

Para ejemplificarlo vamos a replantear el desarrollo de los algoritmos anteriores y reemplazaremos la estructura “**ir a**” (*goto*) por una estructura iterativa: la estructura “**repetir mientras que**”.



A Corrado Böhm y Giuseppe Jacopini se les atribuye el teorema de la programación estructurada, debido a un artículo de 1966.
David Harel rastreó los orígenes de la programación estructurada hasta la descripción de 1946 de la arquitectura de von Neumann y el teorema de la forma normal de Kleene.
La carta escrita en 1968 por Dijkstra, titulada "*Go To Considered Harmful*" reavivó el debate.
En la Web de apoyo, encontrará el vínculo a la página Web personal de Corrado Böhm.

Veamos el desarrollo del módulo `esperarSemaforo`:

Con "**ir a**"	Sin ~~"**ir a**"~~ (solución correcta)
• `observarLuz`; • **Si** la luz está en "rojo" o en "verde intermitente" **entonces** - `esperar`; - **ir a**: `observarLuz`;	• `observarLuz`; • **Repetir mientras que** la luz esté en "rojo" o en "verde intermitente" **hacer** - `esperar`; - `observarLuz`;

Veamos ahora el desarrollo del módulo `cruzarCalle`:

Con "**ir a**"	Sin ~~"**ir a**"~~ (solución correcta)
• `bajarCalzada`; • `avanzarPaso`; • **Si** la distancia hacia la otra calzada es mayor que la distancia que podemos avanzar dando un nuevo paso **entonces** - **ir a**: `avanzarPaso`; • `subirCalzada`;	• `bajarCalzada`; • `avanzarPaso`; • **Repetir mientras que** la distancia hacia la otra calzada sea mayor que la distancia que podemos avanzar dando un nuevo paso **hacer** - `avanzarPaso`; • `subirCalzada`;

Como vemos, la combinación "acción condicional + *goto*" se puede reemplazar por una estructura de repetición. Si bien ambos desarrollos son equivalentes la nueva versión es mejor porque evita el uso del *goto*, estructura que quedó en desuso porque trae grandes problemas de mantenibilidad.

1.3 Conceptos de programación

En general, estudiamos algoritmos para aplicarlos a la resolución de problemas mediante el uso de la computadora. Las computadoras tienen memoria y tienen la capacidad de resolver operaciones aritméticas y lógicas. Por lo tanto, son la herramienta fundamental para ejecutar los algoritmos que vamos a desarrollar.

Para que una computadora pueda ejecutar las acciones que componen un algoritmo tendremos que especificarlas o describirlas de forma tal que las pueda comprender.

Todos escuchamos alguna vez que "las computadoras solo entienden 1 (unos) y 0 (ceros)" y, efectivamente, esto es así, pero para nosotros (simples humanos) especificar las acciones de nuestros algoritmos como diferentes combinaciones de unos y ceros sería una tarea verdaderamente difícil. Afortunadamente, existen los lenguajes de programación que proveen una solución a este problema.



Los lenguajes naturales son los que hablamos y escribimos los seres humanos: inglés, español, italiano, etc. Son dinámicos ya que, constantemente, incorporan nuevas variaciones, palabras y significados. Por el contrario, los lenguajes formales son rígidos y se construyen a partir de un conjunto de símbolos (alfabeto) unidos por un conjunto de reglas (gramática). Los lenguajes de programación son lenguajes formales.

1.3.1 Lenguajes de programación

Las computadoras entienden el lenguaje binario (unos y ceros) y nosotros, los humanos, entendemos lenguajes naturales (español, inglés, portugués, etc.).

Los lenguajes de programación son lenguajes formales que se componen de un conjunto de palabras, generalmente en inglés, y reglas sintácticas y semánticas.

Podemos utilizar un lenguaje de programación para escribir o codificar nuestro algoritmo y luego, con un programa especial llamado "compilador", podremos generar los "unos y ceros" que representan sus acciones. De esta manera, la computadora será capaz de comprender y convertir al algoritmo en un programa de computación.



El lenguaje C++ fue creado a mediados de los años ochenta por Bjarne Stroustrup, con el objetivo de extender al lenguaje de programación C con mecanismos que permitan la manipulación de objetos.



Java es un lenguaje de programación orientado a objetos, creado en la década del noventa por Sun Microsystems (actualmente adquirida por Oracle). Utiliza gran parte de la sintaxis de C y C++, pero su modelo de objetos es más simple.



El lenguaje de programación C fue creado por Dennis Ritchie (1941-2011). En 1967 Ritchie comenzó a trabajar para los laboratorios Bell, donde se ocupó, entre otros, del desarrollo del lenguaje B. Creó el lenguaje de programación C en 1972, junto con Ken Thompson.
Ritchie también participó en el desarrollo del sistema operativo Unix.

Existen muchos lenguajes de programación: Pascal, C, Java, COBOL, Basic, Smalltalk, etc., y también existen muchos lenguajes derivados de los anteriores: Delphi (derivado de Pascal), C++ (derivado de C), C# (derivado de C++), Visual Basic (derivado de Basic), etcétera.

En este libro utilizaremos el lenguaje de programación C y, para los capítulos de encapsulamiento y programación orientada a objetos, C++ y Java.

1.3.2 Codificación de un algoritmo

Cuando escribimos las acciones de un algoritmo en algún lenguaje de programación decimos que lo estamos "codificando". Generalmente, cada acción se codifica en una línea de código.

Al conjunto de líneas de código, que obtenemos luego de codificar el algoritmo, lo llamamos "código fuente".

El código fuente debe estar contenido en un archivo de texto cuyo nombre debe tener una extensión determinada que dependerá del lenguaje de programación que hayamos utilizado. Por ejemplo, si codificamos en Pascal entonces el nombre del archivo debe tener la extensión ".pas". Si codificamos en Java entonces la extensión del nombre del archivo deberá ser ".java" y si el algoritmo fue codificado en C entonces el nombre del archivo deberá tener la extensión ".c".

1.3.3 Bibliotecas de funciones

Los lenguajes de programación proveen bibliotecas o conjuntos de funciones a través de las cuales se ofrece cierta funcionalidad básica que permite, por ejemplo, leer y escribir datos sobre cualquier dispositivo de entrada/salida como podría ser la consola.

Es decir, gracias a que los lenguajes proveen estos conjuntos de funciones los programadores estamos exentos de programarlas y simplemente nos limitaremos a utilizarlas.

Programando en C, por ejemplo, cuando necesitemos mostrar un mensaje en la pantalla utilizaremos la función `printf` y cuando queramos leer datos a través del teclado utilizaremos la función `scanf`. Ambas funciones forman parte de la biblioteca estándar de entrada/salida de C, también conocida como "stdio.h".

1.3.4 Programas de computación

Un programa es un algoritmo que ha sido codificado y compilado y que, por lo tanto, puede ser ejecutado en una computadora.

El algoritmo constituye la lógica del programa. El programa se limita a ejecutar cada una de las acciones del algoritmo. Por esto, si el algoritmo tiene algún error entonces el programa también lo tendrá y si el algoritmo es lógicamente perfecto entonces el programa también lo será.

El proceso para crear un nuevo programa es el siguiente:

- Diseñar y desarrollar su algoritmo.
- Codificar el algoritmo utilizando un lenguaje de programación.
- Compilarlo para obtener el código de máquina o archivo ejecutable (los "unos y ceros").

Dado que el algoritmo es la parte lógica del programa muchas veces utilizaremos ambos términos como sinónimos ya que para hacer un programa primero necesitaremos diseñar su algoritmo. Por otro lado, si desarrollamos un algoritmo seguramente será para, luego, codificarlo y compilarlo, es decir, programarlo.

1.3.5 Consola

Llamamos “consola” al conjunto compuesto por el teclado y la pantalla de la computadora en modo texto. Cuando hablemos de ingreso de datos por consola nos estaremos refiriendo al teclado y cuando hablemos de mostrar datos por consola estaremos hablando de la pantalla, siempre en modo texto.

1.3.6 Entrada y salida de datos

Llamamos “entrada” al conjunto de datos externos que ingresan al algoritmo. Por ejemplo, el ingreso de datos por teclado, la información que se lee a través de algún dispositivo como podría ser un lector de código de barras, un *scanner* de huellas digitales, etcétera.

Llamamos “salida” a la información que el algoritmo emite sobre algún dispositivo como ser la consola, una impresora, un archivo, etcétera.

La consola es el dispositivo de entrada y salida por omisión. Es decir, si hablamos de ingreso de datos y no especificamos nada más será porque el ingreso de datos lo haremos a través del teclado. Análogamente, si hablamos de mostrar cierta información y no especificamos más detalles nos estaremos refiriendo a mostrarla por la pantalla de la computadora, en modo texto.

1.3.7 Lenguajes algorítmicos

Llamamos “lenguaje algorítmico” a todo recurso que permita describir con mayor o menor nivel de detalle los pasos que componen un algoritmo.

Los lenguajes de programación, por ejemplo, son lenguajes algorítmicos ya que, como veremos más adelante, la codificación del algoritmo es en si misma una descripción detallada de los pasos que lo componen. Sin embargo, para describir un algoritmo no siempre será necesario llegar al nivel de detalle que implica codificarlo. Una opción válida es utilizar un “pseudocódigo”.

1.3.8 Pseudocódigo

El pseudocódigo surge de mezclar un lenguaje natural (por ejemplo, el español) con ciertas convenciones sintácticas y semánticas propias de un lenguaje de programación. Justamente, el algoritmo que resuelve el problema de “cruzar la calle” fue desarrollado utilizando un pseudocódigo.

Los diagramas también permiten detallar los pasos que componen un algoritmo; por lo tanto, son lenguajes algorítmicos. En este libro profundizaremos en el uso de diagramas para luego codificarlos con el lenguaje de programación C.



Los diagramas Nassi-Shneiderman, también conocidos como "diagramas de Chapin", fueron publicados en 1973 por Ben Shneiderman e Isaac Nassi en el artículo llamado "A short history of strutctured flowcharts", con el objetivo de eliminar las líneas de los diagramas tradicionales y así reforzar la idea de estructuras de “única entrada y única salida”.
Si en la programación estructurada se elimina la sentencia GOTO entonces se deben eliminar las líneas en los diagramas que la representan.

1.4 Representación gráfica de algoritmos

Los algoritmos pueden representarse mediante el uso de diagramas. Los diagramas proveen una visión simplificada de la lógica del algoritmo y son una herramienta importantísima que utilizaremos para analizar y documentar los algoritmos o programas que vamos a desarrollar.

En este libro, utilizaremos una versión modificada de los diagramas de *Nassi-Shneiderman*, también conocidos como diagramas *Chapin*. Estos diagramas se componen de un conjunto de símbolos que permiten representar cada una de las estructuras de control que describe el teorema de la programación estructurada: la estructura secuencial, la estructura de decisión y la estructura de repetición.

1.4.1 Representación gráfica de la estructura secuencial o acción simple

La estructura secuencial se representa en un recuadro o, si se trata de un ingreso o egreso de datos se utiliza un trapecio como observamos a continuación:

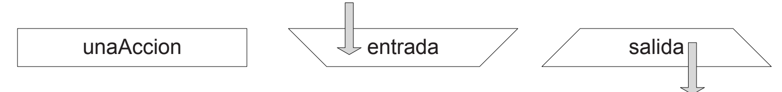


Fig. 1.1 Representación de estructuras secuenciales.

Nota: las flechas grises no son parte de los símbolos de entrada y salida, simplemente son ilustrativas.

1.4.2 Representación gráfica de la estructura de decisión

Esta estructura decide entre ejecutar un conjunto de acciones u otro en función de que se cumpla o no una determinada condición o expresión lógica.

Informalmente, diremos que una "expresión lógica" es un enunciado susceptible de ser verdadero o falso. Por ejemplo, "2 es par" o "5 es mayor que 8". Ambas expresiones tienen valor de verdad, la primera es verdadera y la segunda es falsa.

La estructura se representa dentro de una "casa con dos habitaciones y techo a dos aguas". En "el altillo" indicamos la expresión lógica que la estructura debe evaluar. Si esta expresión resulta ser verdadera entonces se ejecutarán las acciones ubicadas en la sección izquierda. En cambio, si la expresión resulta ser falsa se ejecutarán las acciones que estén ubicadas en la sección derecha.

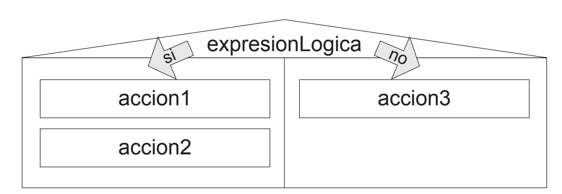


Fig. 1.2 Representación gráfica de la estructura condicional.

En este caso, si se verifica `expresionLogica` se ejecutarán las acciones `accion1` y `accion2`. En cambio, si `expresionLogica` resulta ser falsa se ejecutará únicamente la acción `accion3`. Las flechas grises son ilustrativas y no son parte del diagrama.

1.4.3 Representación gráfica de la estructura de repetición

La estructura de repetición se representa en una "caja" con una cabecera que indica la expresión lógica que se debe cumplir para seguir ejecutando las acciones contenidas en la sección principal.

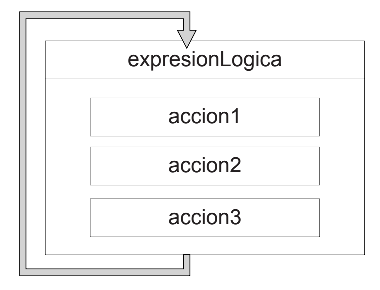


Fig. 1.3 Representación gráfica de la estructura de repetición.

En este caso, mientras `expresionLogica` resulte ser verdadera se repetirán una y otra vez las acciones `accion1`, `accion2` y `accion3`. Por lo tanto, dentro de alguna de estas acciones deberá suceder algo que haga que la condición del ciclo se deje de cumplir. De lo contrario, el algoritmo se quedará iterando dentro de este ciclo de repeticiones.

Con lo estudiado hasta aquí, podemos representar gráficamente el algoritmo que resuelve el problema de cruzar la calle.

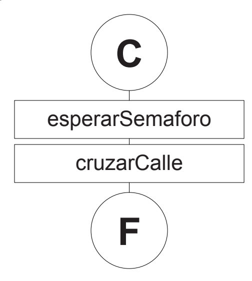


Fig. 1.4 Diagrama del algoritmo principal.

El diagrama principal comienza con una **C** y finaliza con una **F**, iniciales de "Comienzo" y "Fin", cada una encerrada dentro de un círculo. Luego, las acciones simples `esperarSemaforo` y `cruzarCalle` se representan dentro de rectángulos, una detrás de la otra. Como cada una de estas acciones corresponde a un módulo tendremos que proveer también sus propios diagramas.

1.4.4 Representación gráfica de módulos o funciones

Como estudiamos anteriormente, un módulo representa un algoritmo que resuelve un problema específico y puntual. Para representar, gráficamente, las acciones que componen a un módulo, utilizaremos un paralelogramo con el nombre del módulo como encabezado y una **R** (inicial de "Retorno") encerrada dentro de un círculo para indicar que el módulo finalizó y que se retornará el control al algoritmo o programa principal.

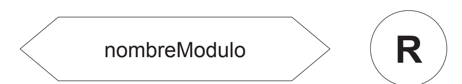


Fig. 1.5 Encabezado y finalización de un módulo.

Con esto, podemos desarrollar los diagramas de los módulos `esperarSemaforo` y `cruzarCalle`.

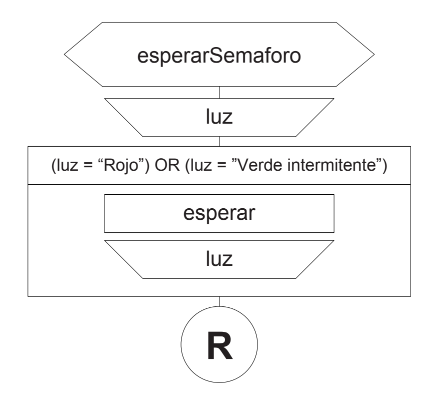


Fig. 1.6 Módulo `esperarSemaforo`.

En el diagrama del módulo `esperarSemaforo`, leemos el color de la luz que actualmente está emitiendo el semáforo y lo "mantenemos en memoria" asociándolo al identificador `luz`. Esto lo hacemos indicando el nombre del identificador dentro del símbolo de lectura o ingreso de datos.

Luego ingresamos a una estructura de repetición que iterará mientras que la expresión lógica `(luz = "Rojo") OR (luz = "Verde intermitente")` resulte verdadera. Las acciones encerradas dentro de esta estructura se repetirán una y otra vez mientras la condición anterior continúe verificándose. Estas acciones son: `esperar` y volver a leer la luz que emite el semáforo.

Evidentemente, en algún momento, el semáforo emitirá la luz "verde fija", la condición de la cabecera ya no se verificará y el ciclo de repeticiones dejará de iterar.

Veamos ahora el diagrama del módulo `cruzarCalle`.

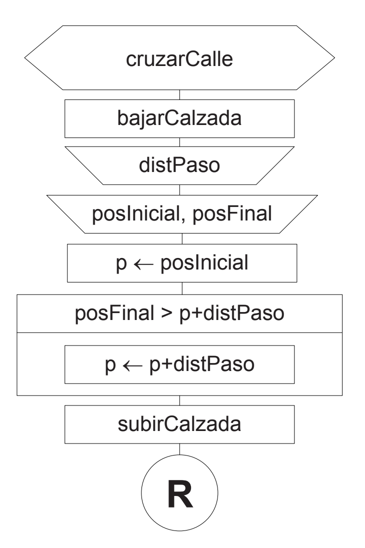


Fig. 1.7 Módulo `cruzarCalle`.

El análisis de este módulo es similar al anterior. Aquí primero invocamos al módulo `bajarCalzada` y luego, ingresamos los datos `distPaso` (distancia que avanzamos al dar un nuevo paso), `posInicial` y `posFinal` (la posición donde estamos parados y la posición de la vereda de enfrente respectivamente). A continuación, "asignamos" al identificador `p` el valor de `posInicial` y luego, ingresamos a una estructura de repetición que iterará mientras que se verifique la expresión lógica:

```
posFinal > p+distPaso
```

Dentro de la estructura de repetición, tenemos que avanzar un paso. Si bien en el primer análisis del algoritmo habíamos definido un módulo `avanzarPaso`, en este caso, preferí o reemplazarlo directamente por la acción:

p ← p+distPaso

Fig. 1.8 Asignación y acumulador.

Esta acción indica que a `p` (identificador que contiene nuestra posición) le asignamos la suma de su valor actual más el valor de `distPaso` (distancia que avanzamos dando un nuevo paso). Luego de esto estaremos más cerca de `posFinal`.

Para finalizar invocamos al módulo `subirCalzada` y retornamos al diagrama del programa principal.

1.5 Nuestro primer programa

Vamos a analizar, diseñar y programar nuestro primer algoritmo. Se trata de un programa trivial que simplemente emite en la consola la frase “Hola Mundo”.

La representación gráfica de este algoritmo es la siguiente:

El algoritmo comienza, emite un mensaje en la consola diciendo “Hola Mundo” y finaliza.

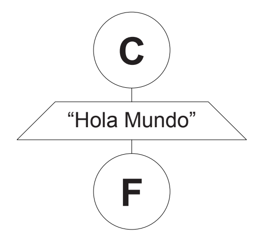


Fig. 1.9 Representación gráfica del algoritmo “Hola Mundo”.

1.5.1 Codificación del algoritmo utilizando el lenguaje C

El siguiente paso es codificar el algoritmo. Con el diagrama resuelto, la codificación se limita a transcribirlo en el `lenguaje` de programación elegido que, en este caso, es C.

Veamos el código fuente y luego lo analizaremos:

```
#include <stdio.h>

int main()
{
   printf("Hola Mundo\n");
   return 0;
}
```

Todos los programas C se codifican dentro de la función `main`. La palabra “*main*” (que significa “principal”) es una palabra reservada y se utiliza como encabezado del programa principal.

Para emitir el mensaje “Hola Mundo” en la consola, utilizamos la función `printf`. Como comentamos anteriormente esta función es parte de la biblioteca estándar de entrada/salida de C llamada “stdio.h”.

La función `printf` recibe como argumento una cadena de caracteres y la imprime en la consola. Notemos que al texto que queremos imprimir le agregamos el carácter especial `\n` (léase “barra ene”). Este carácter representa un salto de línea.

Prácticamente, todos los programas C comienzan con la directiva (de preprocesador):

```
#include <stdio.h>
```

Esta directiva “incluye” las definiciones de las funciones de entrada y salida de la biblioteca estándar.



El preprocesador es un programa que es invocado por el compilador antes de comenzar su trabajo para, por ejemplo, eliminar los comentarios y otros elementos no necesarios para la compilación.



El valor de retorno de la función `main`, también llamado *exit code*, indica como finalizó el programa. Un valor igual a 0 representa una finalización exitosa mientras que un valor diferente indica algún grado de anormalidad.
El *exit code* se utiliza en la programación de *scripts* ya que en función de este se puede determinar si el programa concretó su tarea o no.

Los bloques de código se delimitan entre `{` y `}` ("llave de apertura" y "llave de cierre"). En este caso, el único bloque de código es el bloque que delimita las líneas que forman parte de la función `main` o programa principal.

La sintaxis del lenguaje de programación C exige que todas las líneas de código (salvo las directivas de preprocesador como `include` y los delimitadores de bloques) finalicen con `;` (punto y coma).

Por último, con la sentencia `return` hacemos que la función retorne (o devuelva) un valor a quién la invocó. Este valor es conocido como "el valor de retorno de la función" y lo analizaremos en detalle en el Capítulo 3.

1.5.2 El archivo de código fuente

La codificación del algoritmo constituye el "código fuente". Estas líneas de código deben estar contenidas en un archivo de texto cuyo nombre tiene que tener la extensión ".c". En este caso, el nombre del archivo de código fuente podría ser: `HolaMundo.c`.

1.5.3 Comentarios en el código fuente

A medida que la complejidad del algoritmo se incremente, el código fuente que tendremos que escribir para codificarlo será más complejo y, por lo tanto, más difícil de entender. Por este motivo, los lenguajes de programación permiten agregar comentarios de forma tal que el programador pueda anotar acotaciones que ayuden a hacer más legible el programa que desarrolló.

En C podemos utilizar dos tipos de comentarios:

- Comentarios en línea: cualquier línea de código que comience con "doble barra" será considerada como un comentario.

  ```
  // esto es un comentario en linea
  ```

- Comentarios en varias líneas: son aquellos que están encerrados entre un "barra asterisco" y un "asterisco barra" como se muestra a continuación:

  ```
  /* Esto es un
  comentario en varias
  lineas de codigo */
  ```

A continuación, agregaremos algunos comentarios al programa "HolaMundo".

```
/*
   Programa: HolaMundo.c
   Autor: Pablo Sznajdleder
   Fecha: 24/junio/2012
*/

// incluye las definiciones de las funciones de la biblioteca estandar
#include <stdio.h>

// programa o funcion principal
int main()
{
   // escribe un mensaje en la consola
   printf("Hola Mundo\n");
   return 0;
}
```

Los comentarios dentro del código fuente ayudan a que el programa sea más legible para el programador y, obviamente, no son tomados en cuenta por el compilador.

1.5.4 La compilación y el programa ejecutable

El próximo paso será compilar el código fuente para obtener el programa ejecutable y poder ejecutarlo (correrlo) en la computadora.

La compilación se realiza con un programa especial llamado “compilador”. Existen diversos compiladores que permiten compilar programas escritos en C. En este libro utilizaremos el compilador *GCC* o *MinGW* (http://www.mingw.org/).

Para compilar el programa `HolaMundo.c` con *GCC* escribimos el siguiente comando en la línea de comandos:

```
gcc HolaMundo.c -o HolaMundo.exe
```

Luego de esto se generará, dentro de la misma carpeta en la que compilamos el programa, un nuevo archivo llamado `HolaMundo.exe` que al ejecutarlo mostrará el texto “Hola Mundo” en la consola.

1.5.5 El entorno integrado de desarrollo (IDE)

Cuando los algoritmos adquieren mayor nivel de complejidad, su codificación y compilación pueden convertirse en verdaderos problemas.

Una IDE (por sus siglas en inglés, *Integrated Development Environment*) es una herramienta que integra todos los recursos que necesita un programador para codificar, compilar, depurar, ejecutar y documentar sus programas.

Para programar en C existen diferentes IDEs. Algunas de estas son: Visual Studio (de Microsoft), C++ Builder (de Borland), Dev C (*open source*), Eclipse (*open source*), etcétera.

En este libro utilizaremos Eclipse. Su instalación, configuración y funcionamiento lo analizaremos en los videotutoriales que lo acompañan. Sin embargo, y a modo de ejemplo, a continuación haremos un breve repaso por la herramienta.



Instalación y uso de Eclipse para C

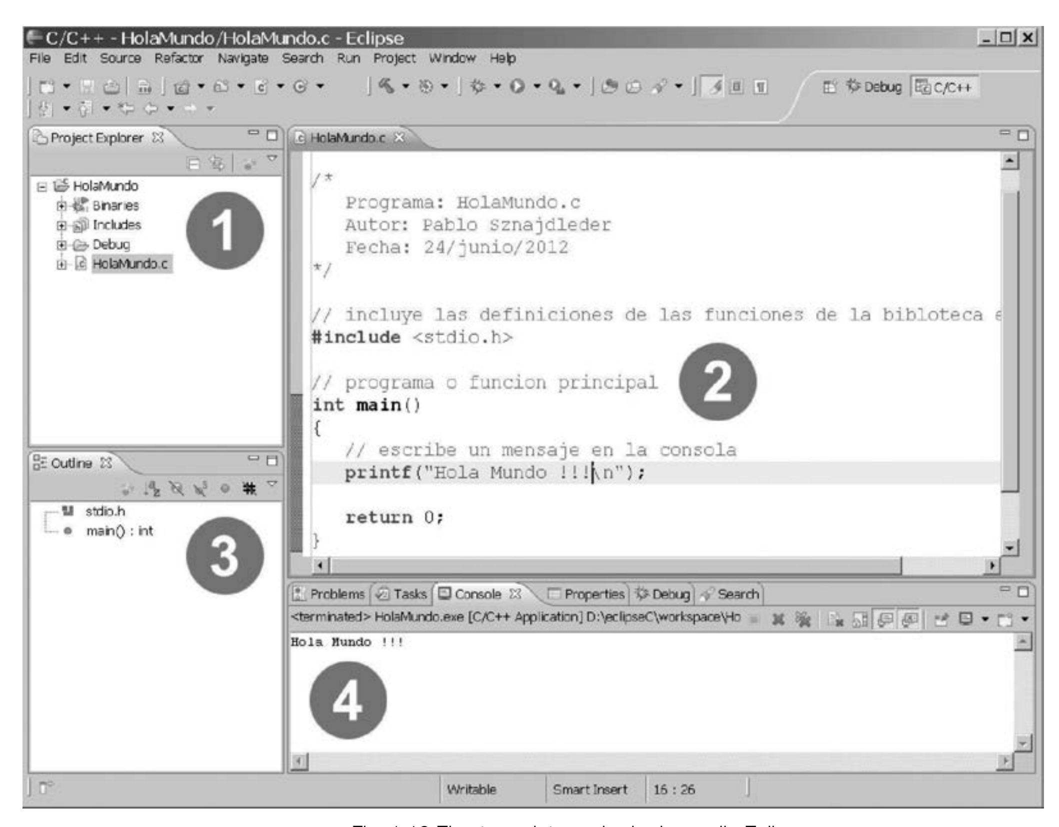


Fig. 1.10 El entorno integrado de desarrollo Eclipse.

En la imagen podemos identificar 4 secciones:

- El **Project Explorer** (1) – Aquí vemos todos los archivos de código fuente que integran nuestro proyecto. En este caso, el proyecto tiene un único archivo: `HolaMundo.c`.
- La **Ventana de Edición** (2) – Es el área principal en donde podemos ver y editar cada uno de los archivos de código fuente.
- El **Outline** (3) – Aquí se detalla la estructura del archivo de código fuente que estamos editando. Se muestra un resumen de todas sus funciones, variables, los `.h` que incluye, etcétera.
- La **Consola** (4) – Es el área en donde ingresaremos los datos a través del teclado y donde se mostrarán los resultados.

1.6 La memoria de la computadora

Más arriba hablamos de entrada y salida de datos. Los datos ingresan al algoritmo a través de cualquier dispositivo de entrada y es nuestra responsabilidad mantenerlos en memoria para tenerlos disponibles y, llegado el momento, poderlos utilizar.

Dependiendo del contenido del dato, necesitaremos utilizar una mayor o menor cantidad de memoria para almacenarlo. Por ejemplo, si el dato corresponde al nombre de una persona probablemente 20 caracteres sean más que suficiente. En cambio, si el dato corresponde a su dirección postal, seguramente, necesitemos utilizar una mayor cantidad de caracteres, quizás 150 o 200.

1.6.1 El byte

La memoria se mide en *bytes*. Un *byte* constituye la mínima unidad de información que podemos almacenar en la memoria. En la actualidad, las computadoras traen grandes cantidades de memoria expresadas en múltiplos del *byte*. Por ejemplo, 1 *gigabyte* representa 1024 *megabytes*. 1 *megabyte* representa 1024 *kilobytes* y un *kilobyte* representa 1024 *bytes*.

A su vez, un *byte* representa un conjunto de 8 bits (dígitos binarios). Por lo tanto, en un *byte* podemos almacenar un número binario de hasta 8 dígitos.

Obviamente, aunque la representación interna del número sea binaria, nosotros podemos pensarlo en base 10 como veremos a continuación.



Bit es el acrónimo de *binary digit*. (dígito binario). Corresponde a un dígito del sistema de numeración binaria que puede representar uno de dos valores, 0 o 1. Es la unidad mínima de información usada en informática.

1.6.2 Conversión numérica: de base 2 a base 10

Un número entero representado en base 2 puede ser fácilmente representado en base 10 realizando los siguientes pasos:

1. Considerar que cada dígito binario representa una potencia de 2, comenzando desde 2° y finalizando en 2^{n-1} siendo *n* la cantidad de dígitos binarios con los que el número está siendo representado.
2. El número en base 10 se obtiene sumando aquellas potencias de 2 cuyo dígito binario sea 1.

Analicemos un ejemplo:

Sea el número binario: 10110011 entonces:

1	0	**1**	**1**	0	0	**1**	**1**
$\mathbf{2^7}$	2^6	$\mathbf{2^5}$	$\mathbf{2^4}$	2^3	2^2	$\mathbf{2^1}$	$\mathbf{2^0}$
128	64	**32**	**16**	8	4	**2**	**1**

Vemos que cada dígito binario representa una potencia de 2, desde la potencia 0 hasta la potencia 7, comenzando desde la derecha.

Concretamente, la representación decimal del número binario 10110011 se obtiene sumando aquellas potencias de 2 cuyo dígito binario es 1.

En este caso será: $2^0+2^1+2^4+2^5+2^7 = 179$.

Según lo anterior, el mayor valor numérico que podremos almacenar en 1 *byte* de información será: 11111111 (8 bits todos en 1) y corresponde al número decimal 255 (que coincide con 2^8-1).

1.6.3 Dimensionamiento de los datos

El análisis anterior es fundamental para comprender que si, por ejemplo, el dato que va a ingresar a nuestro algoritmo corresponde a la edad de una persona podremos almacenarlo sin problemas en 1 *byte* de memoria ya que, salvo Matusalén, nadie llega a vivir 255 años.



En el Antiguo Testamento, se menciona a Matusalén como la persona más anciana. Vivió hasta los 969 años según Génesis 5:27.

También podríamos almacenar en 1 *byte* datos tales como:

- El día del mes (un número que puede estar entre 1 y 31)
- La nota obtenida en un examen (número entre 1 y 10 o entre 1 y 100)
- El número que salió en la ruleta (número entre 0 y 36)

En cambio, no podemos almacenar en 1 *byte*: el día del año ya que este valor será un número entre 1 y 365. Y no sería prudente almacenar en 1 *byte* la distancia (expresada en kilómetros) que existe entre dos ciudades ya que muy probablemente esta distancia sea superior a los 255 km.

Para estos casos necesitaremos utilizar 2 *bytes* de memoria. En 16 bits podremos almacenar valores numéricos hasta $2^{16}-1 = 65535$.

1.6.4 Los números negativos

En el párrafo anterior, vimos que en 1 *byte* podemos representar valores numéricos enteros entre 0 y 255, en 2 *bytes* podemos representar valores numéricos enteros entre 0 y 65535 y, en general, en *n bytes* podremos representar valores numéricos enteros entre 0 y 2^n-1.

Para representar valores numéricos enteros negativos, se reserva el bit más significativo del conjunto de *bytes* para considerarlo como "bit de signo". Si este bit vale 1 entonces los restantes bits del conjunto de *bytes* representarán un valor numérico entero negativo. En cambio, si este bit vale 0 entonces los bits restantes estarán representando un valor numérico entero positivo.

Así, en un *byte* con bit de signo, el mayor valor que se podrá representar será 01111111 (un cero y siete unos) y corresponde al valor decimal 127 (es decir: 2^7-1) y el menor valor que se podrá almacenar será 10000000 (un 1 y siete 0) y corresponde al valor decimal: -127 (*).

Análogamente, en 2 *bytes* con bit de signo el mayor valor que podremos representar será: 0111111111111111 (un cero y 15 unos), que corresponde al valor decimal 32767 mientras que el menor valor será -32767 (*).

Ahora bien, si ***0***0000000 representa al valor decimal 0 (cero) entonces ¿la combinación ***1***0000000 representa a -0 ("menos cero")?

La respuesta a este planteamiento se fundamenta en la representación binaria interna que utilizan las computadoras para los números negativos y que se basa en el complemento a 2 del número positivo. En este esquema los valores negativos se representan invirtiendo los bits que representan a los números positivos y luego, sumando 1 al resultado.

La operación inversa nos permite obtener el valor absoluto de cualquier número binario que comience con 1. Así, el valor absoluto del número 10000000 (-0) será:
01111111+1 = 10000000 = **128**.

Por este motivo, los tipos de datos que representan números enteros signados admiten valores entre -2^{n-1} y $+2^{n-1}-1$ siendo *n* la cantidad de bits utilizados en dicha representación. 1 *byte* implica *n*=8, 2 *bytes* implica *n*=16, etcétera.

1.6.5 Los caracteres

Los caracteres se representan como valores numéricos enteros positivos. Cada carácter tiene asignado un valor numérico definido en la tabla ASCII. En esta tabla se define que el carácter 'A' se representa con el valor 65, el carácter 'B' con el 66 y así sucesivamente. Para representar al carácter 'a' se utiliza el valor 97, el 'b' se representa con el valor 98, el carácter '0' con el valor 48, el '1' con el 49, etcétera.

Por lo tanto, para representar cada carácter alcanzará con 1 *byte* de memoria y *n* caracteres requerirán *n bytes* de memoria.

1.7 Las variables

Las variables representan un espacio de la memoria de la computadora. A través de una variable, podemos almacenar temporalmente datos para tenerlos disponibles durante la ejecución del programa.

Para utilizar una variable, tenemos que especificar un nombre y un tipo de datos. Al nombre de la variable lo llamamos "identificador". En general, un identificador debe comenzar con una letra y no puede contener espacios en blanco, signos de puntuación u operadores.

Nombres de variables o identificadores **válidos** son:

- `fecha`
- `fec`
- `fechaNacimiento`
- `fechaNac`
- `fec1`
- `fec2`
- `iFechaNac`

Nombres de variables o identificadores **incorrectos** son:

- `2fecha` (no puede comenzar con un número)
- `-fecha` (no puede comenzar con un signo "menos")
- `fecha nacimiento` (no puede tener espacios en blanco)
- `fecha-nacimiento` (no puede tener el signo "menos")
- `fecha+nacimiento` (no puede tener el signo "más")

Los valores que mantienen las variables pueden cambiar durante la ejecución del programa, justamente por eso, son "variables". Para que una variable adquiera un determinado valor se lo tendremos que asignar manualmente con el operador de asignación o bien leer un valor a través de algún dispositivo de entrada y almacenarlo en la variable. Esto lo representaremos gráficamente de la siguiente manera:

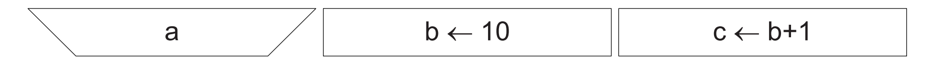


Fig. 1.11 Representación gráfica de lectura y asignación.

En las figuras vemos representadas, de izquierda a derecha, las siguientes acciones:

- Leo un valor por consola y lo asigno a la variable `a`.
- Asigno el valor 10 a la variable `b`.
- Asigno a la variable `c` el valor que contiene la variable `b` "más 1".

Recordemos que para definir una variable es necesario especificar su tipo de datos lo que significa que una variable solo podrá contener datos del mismo tipo. Es decir, la variable puede tomar diferentes valores durante la ejecución del programa, pero todos estos valores siempre serán del mismo tipo: el tipo de datos con el que la variable fue definida.

1.7.1 Convención de nomenclatura para variables

Si bien el lenguaje de programación solo exige que los nombres de las variables o identificadores respeten las restricciones mencionadas más arriba, en este libro, adoptaremos la siguiente convención de nomenclatura:

Nombres simples: deben escribirse completamente en minúscula. Por ejemplo: `fecha`, `nombre`, `edad`, `direccion`, etcétera.

Nombres compuestos: si el nombre de la variable está compuesto por dos o más palabras entonces cada palabra, a excepción de la primera, debe comenzar en mayúscula. Por ejemplo: `fechaNacimiento`, `nombreYApellido`, etcétera.

Nunca el nombre de una variable debería comenzar en mayúscula ni escribirse completamente en mayúscula.

1.7.2 Los tipos de datos

Dependiendo del tipo de datos que definamos para una variable, esta estará representando una mayor o menor cantidad de *bytes* de memoria y, por lo tanto, nos permitirá almacenar mayor o menor cantidad de información.

Según su contenido, los datos pueden clasificarse como numéricos, alfanuméricos o lógicos.

Por ejemplo: la dirección postal de una persona es un dato **alfanumérico**. Este tipo de dato representa una sucesión de caracteres alfabéticos y/o numéricos como podría ser "Av. Del Libertador Nro. 2345, piso 6".

En cambio, la edad de una persona es un dato numérico, entero positivo y acotado ya que, como analizamos más arriba, 255 es una edad absurda a la que un ser humano nunca podrá llegar. Es decir que este dato es un número **entero corto** y sin bit de signo o *unsigned* ya que ninguna persona podrá tener una edad negativa.

El kilometraje de un auto, por ejemplo, es un valor entero positivo y, potencialmente, muy grande. No podrá representarse en 1 *byte* (255), ni siquiera en 2 *bytes* (65535). Tendremos que utilizar una mayor cantidad de *bytes*. A este tipo de datos lo llamaremos **entero largo**. Este dato también es *unsigned* ya que un auto no puede tener un kilometraje negativo.



El ecuador divide al globo en el hemisferio norte y el hemisferio sur, es una línea imaginaria (un círculo máximo) que se encuentra, exactamente, a la misma distancia de los polos geográficos.

La distancia (expresada en kilómetros) que existe entre dos ciudades es un dato que puede representarse en 2 *bytes*. Esto se desprende del siguiente razonamiento: Como el diámetro del ecuador es de aproximadamente 12710 km entonces el perímetro de la Tierra se puede calcular como π*diámetro = 40074 km, también aproximado. Es decir que, en el peor de los casos, una ciudad puede estar en un punto de la Tierra y la otra puede estar justo al otro lado del mundo y aun así, la distancia nunca será mayor que la mitad de perímetro de la Tierra: 20037 km.

El saldo de una cuenta bancaria es un valor numérico real. Es decir, probablemente tenga decimales y podrá ser positivo o negativo. A este tipo de datos lo llamamos **flotante**. La representación interna de los números con punto flotante la analizaremos más adelante.

Según sea el grado de precisión que demande el valor real que tenemos que representar, su tipo podrá ser simplemente "flotante" o bien flotante de **doble precisión**.

Por último, los datos lógicos o **booleanos** son aquellos que tienen valor de verdad. Este valor puede ser verdadero o falso. En general, utilizamos este tipo de dato para almacenar el resultado de una operación lógica.

1.7.3 Los tipos de datos provistos por el lenguaje C

Los lenguajes de programación proveen tipos de datos con los cuales se puede definir (o declarar) variables.

Como vimos más arriba, para utilizar una variable será necesario definir su identificador (nombre de la variable) y su tipo de datos. En C disponemos de los siguientes tipos:

Naturaleza	Tipo	Bytes	Descripción
Tipos numéricos enteros	*short*	1	entero corto
	int	2	entero
	long	4	entero largo
	char	1	carácter (entero corto)
Tipos numéricos flotantes (o reales)	*float*	4	flotante
	double	8	flotante doble precisión

Nota muy importante: Las cantidades de *bytes* no siempre serán las mencionadas en la tabla ya que estas dependerán del compilador y de la arquitectura del *hardware* en donde estemos compilando y ejecutando nuestro programa. Sin embargo, en este libro, por cuestiones didácticas, consideraremos que esta será la cantidad de *bytes* de memoria reservada para cada tipo de dato.

A cada uno de los tipos de datos enteros se les puede aplicar el modificador `unsigned` con lo que podremos indicar si queremos o no que se deje sin efecto el bit de signo. Con esto, aumentamos el rango de valores positivos que admite el tipo de datos a costa de sacrificar la posibilidad de almacenar valores negativos.

Los datos lógicos o booleanos se manejan como tipos enteros, considerando que el valor 0 es “falso” y cualquier otro valor distinto de 0 es “verdadero”. En otros lenguajes, como Pascal o Java, se provee el tipo de datos `boolean`, pero en C los datos lógicos simplemente se trabajan como `int`.

Por último, los datos alfanuméricos o “cadenas de caracteres” se representan como “conjuntos de variables” de tipo `char`. A estos “conjuntos” se los denomina *arrays*, un tema que estudiaremos en detalle más adelante.

1.7.3.1 Notación húngara

La notación húngara es una convención de nomenclatura que propone anteponer al nombre de la variable un prefijo que, a simple vista, permita identificar su tipo de datos.

Por ejemplo: `iContador` (“i” de `int`), `sNombre` (“s” de *string*), `bFin` (“b” de *boolean*), `dPrecioVenta` (“d” de `double`), etcétera.

Si bien la notación húngara cuenta con una importante comunidad de detractores que aseguran que, a la larga, complica la legibilidad y la mantenibilidad del código fuente, personalmente considero que, en ciertas ocasiones, se puede utilizar.

1.7.4 La función de biblioteca `printf`

Como ya sabemos la función `printf` permite mostrar datos en la consola. La salida puede estar compuesta por un texto fijo (texto literal) o por una combinación de texto literal y contenido variable como veremos en el siguiente programa.

```
#include <stdio.h>

int main()
{
   char nombre[] = "Pablo";
   int edad = 39;
   double altura = 1.70;
```

```
   printf("Mi nombre es %s, tengo %d anios y mido %lf metros.\n"
          ,nombre
          ,edad
          ,altura);

   return 0;
}
```

La salida de este programa es:

```
Mi nombre es Pablo, tengo 39 anios y mido 1.70 metros.
```

La función `printf` intercaló los valores de las variables `nombre`, `edad` y `altura` dentro del texto literal, en las posiciones indicadas con los `%` (marcadores de posición o *placeholders*).

La cadena que contiene el texto literal con los marcadores de posición se llama "máscara" y es el primer argumento que recibe `printf`. A continuación, le pasamos tantos argumentos como *placeholders* tenga la máscara.

```
printf("Mi nombre es %s, tengo %d anios y mido %lf metros.\n"
       ,nombre
       ,edad
       ,altura);
```

Cada marcador debe ir acompañado de un carácter que representa un tipo de datos. Por ejemplo, `%s` indica que allí se mostrará un valor alfanumérico, `%d` representa a un valor entero y `%lf` indica que se mostrará un valor flotante.

Por último, pasamos la lista de variables cuyos valores se intercalarán en el texto que queremos que `printf` escriba en la consola.

1.7.5 La función de biblioteca `scanf`

Esta función permite leer datos a través del teclado y los asigna en las variables que le pasemos como argumento.

Analicemos el código del siguiente programa donde se le pide al usuario que ingrese datos de diferentes tipos.

```
#include <stdio.h>

int main()
{
   char nombre[20];
   int edad;
   double altura;

   printf("Ingrese su nombre: ");
   scanf("%s",nombre);

   printf("Ingrese su edad: ");
   scanf("%d",&edad);

   printf("Ingrese su altura: ");
   scanf("%lf",&altura);
```

```
   printf("Ud. es %s, tiene %d anios y una altura de %lf\n"
         ,nombre
         ,edad
         ,altura);

   return 0;
}
```

`scanf` recibe como primer argumento una máscara que indica el tipo de los datos que la función debe leer. Los marcadores coinciden con los que utiliza `printf`, por tanto con `%s` le indicamos que lea una cadena de caracteres y con `%d` y `%lf` le indicamos que lea un entero y un flotante respectivamente.

Para que una función pueda modificar el valor de un argumento tenemos que pasarle su referencia o su dirección de memoria. Esto lo obtenemos anteponiendo el operador `&` (léase "operador ampersand") al identificador de la variable cuyo valor queremos que la función pueda modificar. Veamos las siguientes líneas:

```
printf("Ingrese su edad: ");
scanf("%d",&edad);

printf("Ingrese su altura: ");
scanf("%lf",&altura);
```

Utilizamos `scanf` para leer un valor entero y asignarlo en la variable `edad`, luego usamos `scanf` para leer un valor flotante y asignarlo en la variable `altura`.

El caso de las cadenas de caracteres es diferente. Como comentamos más arriba, las cadenas se manejan como *arrays* (o conjuntos) de caracteres. Si bien este tema lo estudiaremos más adelante, es importante saber que el identificador de un *array* es en sí mismo su dirección de memoria, razón por la cual no fue necesario anteponer el operador `&` a la variable `nombre` para que `scanf` pueda modificar su valor.

```
printf("Ingrese su nombre: ");
scanf("%s",nombre);
```

Nota: en todos los casos `scanf` lee desde el teclado valores alfanuméricos. Luego, dependiendo de la máscara, convierte estos valores en los tipos de datos que corresponda y los asigna en sus respectivas variables.

1.7.6 El operador de dirección `&`

El operador `&` se llama "operador de dirección" y aplicado a una variable nos permite obtener su referencia o dirección de memoria.

Tendremos que utilizar este operador cada vez que necesitemos que una función pueda modificar el valor de alguna variable que le pasemos como argumento.

1.7.7 Las constantes

Los algoritmos resuelven situaciones problemáticas que surgen de la realidad, donde existen valores que nunca cambiarán. Dos ejemplos típicos son los números `PI` y `E` cuyas aproximaciones son: 3.141592654 y 2.718281828 respectivamente.

1.7.7.1 La directiva de preprocesador `#define`

Esta directiva permite definir valores constantes de la siguiente manera:

```
#define NUMERO_PI 3.1415169254
#define NUMERO_E 2.718281828
```

El preprocesador de C reemplazará cada aparición de `NUMERO_PI` y `NUMERO_E` por sus respectivos y correspondientes valores.

1.7.7.2 El modificador `const`

Si a la declaración de una variable le aplicamos el modificador `const` su valor ya no podrá ser modificado. Por ejemplo:

```
const int temperaturaMaxima = 45;
```

Luego, cualquier intento de asignar otro valor a la variable `temperaturaMaxima` en el código del programa generará un error de compilación.

1.7.8 Nomenclatura para las constantes

Adoptaremos la siguiente convención de nomenclatura para los nombres de las constantes definidas con la directiva `#define`:

Las constantes deben escribirse completamente en mayúscula. Luego, si se trata de un nombre compuesto por dos o más palabras, cada una debe separarse de la anterior mediante el carácter guión bajo o *underscore*.

En cambio, para las constantes declaradas con el modificador `const` respetaremos la convención de nomenclatura para variables estudiada más arriba.

1.8 Operadores aritméticos

Comenzamos el capítulo explicando que los recursos que tenemos para diseñar y desarrollar algoritmos son la memoria y la capacidad de ejecutar operaciones aritméticas y lógicas.

Todos los lenguajes de programación proveen un conjunto de operadores con los que podemos realizar operaciones aritméticas. En C estos operadores son los siguientes:

Operador	Descripción
+	suma
-	resta
*	multiplicación
/	división (cociente)
%	módulo, resto o valor residual

Veamos un ejemplo.

Problema 1.1

Leer dos valores enteros e informar su suma.

Análisis

En este problema, los datos de entrada son los dos valores numéricos enteros que ingresará el usuario. La salida del algoritmo será un mensaje en la consola informando la suma de estos valores. El proceso implica sumar los dos valores y mostrar el resultado en la pantalla.

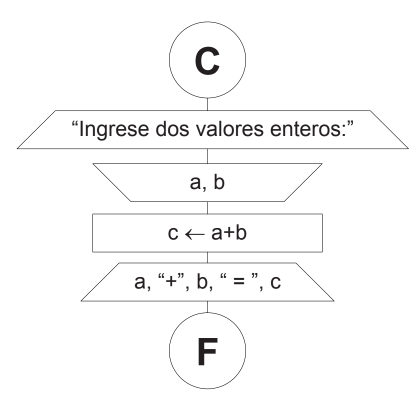


Fig. 1.12 Suma dos valores numéricos.

Comenzamos el algoritmo emitiendo un mensaje para informarle al usuario que debe ingresar dos valores numéricos enteros. A continuación, leemos los valores que el usuario va a ingresar, quedarán asignados en las variables `a` y `b`. Luego asignamos en la variable `c` la suma `a+b` y mostramos el resultado intercalando los valores de las variables `a`, `b` y `c` con las cadenas literales "+" e "=".

La codificación del algoritmo es la siguiente:

```
#include <stdio.h>

int main()
{
   int a,b,c;

   printf("Ingrese dos valores enteros: ");
   scanf("%d %d",&a, &b);

   c = a+b;

   printf("%d + %d = %d\n",a,b,c);

   return 0;
}
```

El algoritmo hubiera sido, prácticamente, el mismo si en lugar de tener que mostrar la suma de los dos valores ingresados por el usuario nos hubiera pedido que mostremos su diferencia o su producto. Pero, ¿qué sucedería si nos pidieran que mostremos su cociente? Lo analizaremos a continuación:

Problema 1.2

Leer dos valores numéricos enteros e informar su cociente.

Análisis

En este problema, los datos de entrada son los dos valores enteros que ingresará el usuario a través del teclado (los llamaremos `a` y `b`) y la salida será su cociente (un número flotante).

Ahora bien, existe la posibilidad de que el usuario ingrese como segundo valor el número 0 (cero). En este caso, no podremos mostrar el cociente ya que la división por cero es una indeterminación, así que tendremos que emitir un mensaje informando las causas por las cuales no se podrá efectuar la operación.

Para resolver este algoritmo utilizaremos una estructura condicional que nos permitirá decidir, en función del valor de `b`, entre mostrar un mensaje de error y realizar la división e informar el cociente. Veamos el algoritmo:

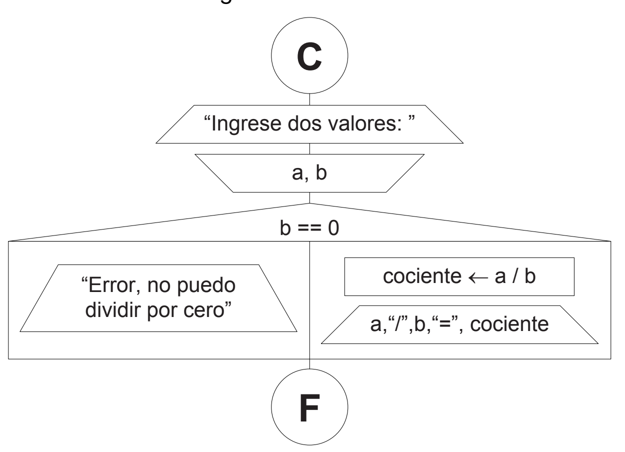


Fig. 1.13 Muestra el cociente de la división de dos números.

Leemos dos valores en las variables `a` y `b` y luego preguntamos si el valor de `b` es cero. Para esto, utilizamos el operador de comparación `==` (léase "igual igual"). Si efectivamente `b` vale cero entonces mostramos un mensaje de error informando lo sucedido, si no asignamos a la variable `cociente` el resultado de la división `a/b` y lo mostramos en la consola. El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   int a,b;
   double cociente;

   printf("Ingrese dos valores: ");
   scanf("%d %d", &a, &b);

   // verifico si el denomicador es cero
   if( b == 0 )
   {
      printf("Error, no puedo dividir por cero\n");
   }
   else
   {
      cociente = (double)a/b;
      printf("%d / %d = %lf\n",a,b,cociente);
   }

   return 0;
}
```

Como vemos, la estructura de decisión se codifica con `if` ("si" en inglés). Si se verifica la condición expresada dentro de los paréntesis del `if` el programa ingresará por el primer bloque de código para mostrar el mensaje de error. Si la condición no se verifica (`else`) el programa ingresará al segundo bloque de código para efectuar la división y mostrar el resultado.

1.8.1 Conversión de tipos de datos (type casting)

C permite convertir datos de un tipo a otro. Esta operación se llama *type casting* o simplemente *casting*.

Si bien la conversión de datos de un tipo a otro es automática, en ocasiones necesitaremos realizarla explícitamente.

Observemos la línea de código donde calculamos la división y asignamos el resultado a la variable `cociente`:

```
cociente = (double)a/b;
```

El operador `/` (operador de división) convierte el resultado al mayor tipo de datos de sus operandos. Esto significa que si estamos dividiendo dos `int` entonces el resultado también será de tipo `int`, por lo tanto, el cociente que obtendremos será el de la división entera.

Para solucionar este problema y no perder los decimales, tenemos que convertir el tipo de alguno de los operandos a `double` (el tipo de datos de la variable `cociente`). Así, el mayor tipo de datos entre `int` y `double` es `double` por lo que el resultado de la división también lo será.

Le recomiendo al lector eliminar el *casteo* y luego recompilar y probar el programa para observar que resultados arroja.

1.8.2 El operador `%` ("módulo" o "resto")

El operador `%` (léase "operador módulo" u "operador resto") retorna el resto (o valor residual) que se obtiene luego de efectuar la división entera de sus operandos.

Por ejemplo:

```
int a = 5;
int b = 3
int r = a % b;
```

En este ejemplo estamos asignando a la variable `r` el valor 2 ya que este es el resto (o valor residual) que se origina al dividir 5 por 3.

Problema 1.3

Dado un valor numérico entero, informar si es par o impar.

Análisis

En este problema tenemos un único dato de entrada: un valor numérico entero que deberá ingresar el usuario. La salida del algoritmo será informar si el usuario ingresó un valor par o impar.

Sabemos que un número par es aquel que es divisible por 2 o, también, que un número es par si el valor residual que se obtiene al dividirlo por 2 es cero.

Según lo anterior, podremos informar que el número ingresado por el usuario es par si al dividirlo por 2 obtenemos un resto igual a cero. De lo contrario, informaremos que el número es impar.

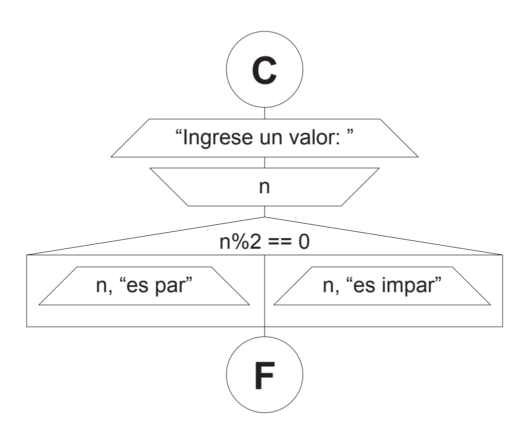


Fig. 1.14 Informa si un número es par o impar.

El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   int n;

   printf("Ingrese un valor: ");
   scanf("%d",&n);

   if( n%2 == 0 )
   {
      printf("%d es par\n",n);
   }
   else
   {
      printf("%d es impar\n",n);
   }

   return 0;
}
```

Veamos otro ejemplo en el cual tendremos que utilizar los operadores aritméticos de módulo y división.

Problema 1.4

Se ingresa un valor numérico de 8 dígitos que representa una fecha con el siguiente formato: *aaaammdd*. Esto es: los 4 primeros dígitos representan el año, los siguientes 2 dígitos representan el mes y los 2 dígitos restantes representan el día. Se pide informar por separado el día, el mes y el año de la fecha ingresada.

Análisis

El dato que ingresará al algoritmo es un valor numérico de 8 dígitos como el siguiente: 20081015. Si este fuera el caso, entonces la salida deberá ser:

```
dia: 15
mes: 10
anio: 2008
```

Es decir, el problema consiste en desmenuzar el número ingresado por el usuario para separar los primeros 4 dígitos, después los siguientes 2 dígitos y luego, los 2 últimos.

Supongamos que efectivamente el valor ingresado es 20081015 entonces si lo dividimos por 10 obtendremos el siguiente resultado:

20081015 / 10 = 2008101,5

Pero si en lugar de dividirlo por 10 lo dividimos por 100 el resultado será:

20081015 / 100 = 200810,15

Siguiendo el mismo razonamiento, si al número lo dividimos por 10000 entonces el resultado que obtendremos será:

20081015 / 10000 = 2008,1015

Si de este valor tomamos solo la parte entera tendremos 2008 que coincide con el año representado en la fecha que ingresó el usuario.

Por otro lado, el resto obtenido en la división anterior será: 1015. Esto coincide con el mes y el día, por lo tanto, aplicando un razonamiento similar podremos separar y mostrar estos valores.

Para representar la "división entera" en los diagramas, utilizaremos el operador `div`. Este operador no existe en C, pero nos permitirá diferenciar, visualmente, entre una división real o flotante y una división entera.

Veamos el algoritmo:

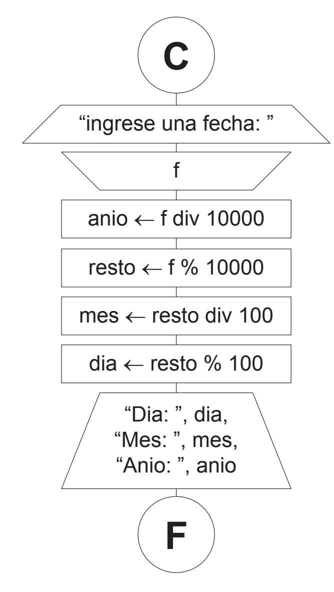


Fig. 1.15 Separa los dígitos de un número entero.

Para codificar este algoritmo debemos tener en cuenta que el valor que ingresará el usuario representa una fecha con formato *aaaammdd*. Es decir que, en el peor de los casos, este valor será 99991231 y no lo podemos almacenar en 2 *bytes* ya que excede por mucho su capacidad (32767 o 65535). Por este motivo, utilizaremos 4 *bytes* que nos permitirán almacenar números de hasta 2^{32}-1 que superan los 8 dígitos, es decir: trabajaremos con variables de tipo `long`.

Recordemos que, por cuestiones didácticas, consideramos que el tipo `int` representa 2 *bytes* y el tipo `long` representa 4 *bytes*.

El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   long f;
   int dia, mes, anio;
   int resto;

   printf("Ingrese una fecha: ");
   scanf("%ld",&f);

   // recordemos que la division entre dos enteros tambien lo sera
   anio = f/10000;
   resto = f%10000;

   mes = resto/100;
   dia = resto%100;

   printf("Dia: %d\n",dia);
   printf("Mes: %d\n",mes);
   printf("Anio: %d\n",anio);

   return 0;
}
```

Recordemos que el operador de división `/` retorna un resultado del mismo tipo de datos que el mayor tipo de sus operandos. Como, en este caso, `f` es de tipo `long` entonces el cociente también lo será. Es decir que la división será entera.

1.8.3 Operadores relacionales

Estos operadores permiten evaluar la relación que existe entre dos valores numéricos. Ya hemos mencionado y utilizado al operador de comparación `==` (igual igual) para comparar si un número es igual a otro. Ahora veremos la lista completa que incluye a los operadores “mayor”, “menor”, “mayor o igual”, “menor o igual” y “distinto”.

Operador	Descripción
>	mayor que...
<	menor que...
>=	mayor o igual que...
<=	menor o igual que...
==	igual a...
!=	distinto de...

Como veremos más adelante, el operador `!` (signo de admiración) es el operador lógico de negación (operador “*not*”), por lo tanto, el operador `!=` puede leerse como “distinto”, “no igual” o “*not equals*”.

1.9 Expresiones lógicas

Llamamos expresión lógica a una proposición que es susceptible de ser verdadera o falsa. Es decir, una proposición que tiene valor de verdad.

Por ejemplo, las siguientes proposiciones son expresiones lógicas cuyo valor de verdad es verdadero:

- La Tierra gira alrededor del Sol.
- EE. UU. tiene dos costas.
- 2 “es menor que” 5 (o simplemente 2<5).
- 5+1 = 6

Y las siguientes proposiciones son expresiones lógicas cuyo valor de verdad es falso:

- La Tierra es el centro del universo.
- EE. UU. queda en Europa.
- 2 “es mayor que” 5 (o simplemente 2>5).
- 5+1 = 8



El Feng Shui explica como nos afectan los flujos energéticos de nuestro entorno, dicha energía es conocida como Chi, circula por los diferentes espacios afectando en su recorrido todo lo que toca.

En cambio, no son expresiones lógicas las siguientes proposiciones:

- Hoy hace frío.
- La pared está bastante sucia.
- Los números impares tienen mejor “Chi”.

Las expresiones lógicas pueden combinarse entre sí formando nuevas expresiones lógicas con su correspondiente valor de verdad. Para esto, se utilizan los operadores lógicos.

1.9.1 Operadores lógicos

Las expresiones lógicas pueden conectarse a través de los operadores lógicos y así se obtienen expresiones lógicas compuestas cuyo valor de verdad dependerá de los valores de verdad de las expresiones lógicas simples que las componen.

Los operadores lógicos son los siguientes:

Operador	**Descripción**
`&&`	“*and*” o producto lógico
`||`	“*or*” o suma lógica
`!`	“*not*” o negación

Con los operadores lógicos podemos conectar dos o más expresiones lógicas y así obtener una nueva expresión lógica con su correspondiente valor de verdad.

Sean las expresiones lógicas `p` y `q` cada una con su correspondiente valor de verdad entonces el valor de verdad de la expresión lógica `h = p && q` será verdadero o falso según la siguiente tabla:

p	**q**	**h = p && q**
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	falso	falso

Es decir, si `p` es verdadero y `q` es verdadero entonces la expresión lógica `p && q` también lo es. En cambio, si alguna de las dos expresiones lógicas (o las dos) es falsa entonces su producto lógico también lo será.

Veamos ahora la tabla del operador `||` (operador "*or*"):

p	**q**	**h = p || q**
verdadero	verdadero	verdadero
verdadero	falso	verdadero
falso	verdadero	verdadero
falso	falso	falso

Como vemos, la suma lógica siempre resulta verdadera a no ser que el valor de verdad de los dos operandos sea falso.

El operador `!` (operador "not") niega el valor de verdad del operando. Es decir, si el operando es verdadero entonces su negación será falsa. En cambio, si el operando es falso su negación será verdadera.

p	**h = !p**
verdadero	falso
falso	verdadero

1.10 Operadores de bits

En este apartado, estudiaremos operadores que permiten manipular los bits de los *bytes* vinculados a las variables enteras que declaramos en los programas.

La complejidad del tema excede a la que venimos manejando en este capítulo introductorio razón por la cual le recomiendo al lector pasar por alto esta lectura ya que, oportunamente, cuando lo considere adecuado le recomendaré retomarla.

1.10.1 Representación binaria de los tipos enteros

Más arriba estudiamos que un valor numérico entero se puede representar en un conjunto de *bytes*. Cuanto más grande sea la cantidad de *bytes* de este conjunto, mayor será la diversidad de valores enteros que permitirá abarcar.

En el lenguaje de programación C, estos conjuntos de *bytes* están representados por los tipos de datos: `char`, `short`, `int` y `long`.

Para facilitar el análisis, nos manejaremos con valores pequeños desarrollando ejemplos basados en el tipo `char` que, como ya sabemos, representa un conjunto de un único *byte* con bit de signo.

Sea la variable `c` definida de la siguiente manera:

```
char c = 38;
```

Internamente, su representación será:

00100110

Luego, si multiplicamos su valor por -1:

```
c = -1*c;
```

Su representación pasará a ser:

***1*1011010**

y la obtenemos al invertir todos los bits del número positivo y luego, sumando 1 al resultado.

1.10.2 Operadores de desplazamiento de bits (>> y <<)

Los operadores de desplazamiento (*shift* en inglés) permiten mover los bits de una variable entera hacia la izquierda (*shift left*) o hacia la derecha (*shift right*).

Continuando con la variable `c` declarada más arriba, si hacemos:

```
// desplazamos los bits de c una posicion hacia la derecha
c = c>>1;
```

`c` pasará a valer 19. Veamos:

c	00100110	38
c = c>>1	00010011	19

Y si en lugar de correr sus bits hacia la derecha los corremos hacia la izquierda:

```
c = c<<1;
```

la variable pasará a valer 76.

c	00100110	38
c = c<<1	01001100	76

¿Qué sucederá si corremos los bits de `c` 2 lugares hacia la izquierda?

c	00100110	38
c = c<<2	10011000	152

El resultado que deberíamos obtener es 152. Sin embargo, como la variable es de tipo `char` solo admite valores positivos no mayores a 127. De esta manera, la asignación desbordará la capacidad de la variable y le asignará un valor absurdo e irreal.

1.10.3 Representación hexadecimal

La representación hexadecimal facilita, enormemente, la tarea de trabajar con bits (dígitos binarios) ya que un solo dígito hexadecimal permite representar 4 dígitos binarios.

La siguiente tabla muestra la combinación binaria que representa cada uno de los 16 dígitos hexadecimales.

Hexadecimal	**Binario**	**Hexadecimal**	**Binario**
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Fig. 1.16 Codificación binaria de los dígitos hexadecimales.

Así, para representar el número binario 01111011 (123 decimal) solo serán necesarios los dígitos hexadecimales: 7B.

Binario:	0111	1011
Hexadecimal:	7	B

En C podemos utilizar el sistema hexadecimal para representar valores enteros. Para esto, tenemos que anteponer el prefijo `0x` (léase “cero equis”) al número hexadecimal.

Las siguientes líneas de código muestran diferentes formas de expresar el mismo valor numérico entero.

```
char a = 123;    // o 01111011 binario
char b = 0x7B;   // o 01111011 binario
```

Ahora las variables `a` y `b` tienen el mismo valor numérico: 123.

Representación hexadecimal de números enteros negativos

La representación hexadecimal de números enteros está separada de la representación binaria interna que el número adquiere cuando está en memoria. Por esto, el número 0x88 (10001000) representa al valor +136 aunque su bit más significativo sea 1. Para representar el valor entero -136, utilizaremos –0x88.

1.10.4 Representación octal

El sistema octal es alternativo al sistema hexadecimal. En este caso, cada dígito octal (0 a 7) representa un conjunto de 3 dígitos binarios.

Octal	Binario	Octal	Binario
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

Fig. 1.17 Codificación binaria de los dígitos octales.

C permite expresar valores octales anteponiendo al número el prefijo 0 (léase “cero”). Luego, necesitaremos 3 dígitos octales para representar el valor numérico entero 123 (01111011).

Binario:	001	111	011
Octal:	1	7	3

```
char x = 123;  // 123 en decimal
char y = 0x7B; // 123 en hexadecimal
char z = 0173; // 123 en octal
```

1.10.5 Operadores lógicos de bits

Sean las siguientes variables declaradas e inicializadas como vemos a continuación:

```
unsigned char a = 0x61; // 01100001
unsigned char b = 0x2D; // 00101101
```

Podemos obtener su producto y su suma lógica de la siguiente manera:

Producto lógico

a	01100001	97
b	00101101	45
c = a&b	00100001	33

Suma lógica

a	01100001	97
b	00101101	45
c = a|b	01101101	109

Fig. 1.18 Operadores lógicos de bits.

El producto lógico entre dos valores enteros se obtiene comparando uno a uno los bits de sus posiciones homónimas. Luego, si ambos bits valen 1 el bit resultante para esa posición también tendrá este valor.

En cambio, para obtener la suma lógica alcanza con que alguno de los bits de una misma posición sea 1 para que el bit resultante también tenga este valor.

1.11 Resumen

En este capítulo, explicamos los conceptos básicos de programación proporcionándoles a los lectores que nunca antes han programado un primer acercamiento a la materia.

Entre estas nociones básicas e introductorias creo conveniente destacar los conceptos de “algoritmo y problema”, las estructuras de control que describe el teorema de la programación estructurada y la idea de variables y tipos de datos.

Además, con la ayuda de los videotutoriales, el lector habrá podido compilar sus primeros programas en el lenguaje C.

En el próximo capítulo, nos concentraremos en estudiar con mayor nivel de detalle las estructuras secuencial, condicional e iterativa, incrementando así el nivel de complejidad de los algoritmos.

1.12 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

1.12.1 Mapa conceptual

1.12.2 Autoevaluaciones

1.12.3 Videotutorial

1.12.3.1 Instalación y uso de Eclipse para C

1.12.4 Presentaciones*

2

Estructuras básicas de control y lógica algorítmica

Contenido

Objetivos del capítulo

- Analizar y resolver algoritmos utilizando las estructuras de control básicas: secuencial, condicional e iterativa.
- Conocer, identificar y aplicar recursos algorítmicos: contadores, acumuladores, máximos, mínimos, etcétera.
- Hacer un seguimiento “paso a paso” del algoritmo y aprender a usar la herramienta de depuración: el *debugger*.
- Analizar y resolver problemas contextualizados.

Competencias específicas

- Dominar los conceptos básicos de la programación.
- Analizar problemas y representar su solución mediante algoritmos.
- Conocer las características principales del lenguaje C.
- Codificar algoritmos en el lenguaje de programación C.
- Compilar y ejecutar programas.
- Construir programas utilizando estructuras condicionales y repetitivas para aumentar su funcionalidad.

2.1 Introducción

En el capítulo anterior, estudiamos el teorema de la programación estructurada que describe tres estructuras básicas de control con las que demuestra que es posible resolver cualquier problema computacional.

En este capítulo, estudiaremos las diferentes implementaciones que provee C para estas estructuras y, además, analizaremos problemas con mayor grado de complejidad que nos permitirán integrar todos los conceptos que incorporamos a lo largo del Capítulo 1.

2.2 Estructura secuencial

En este caso, simplemente, haremos un breve repaso para recordar que la estructura secuencial consiste en ejecutar, secuencialmente, una acción simple detrás de otra.

Recordemos, también, que se considera acción simple a las acciones de leer, escribir, asignar valor a una variable e invocar a un módulo o función.

2.3 Estructura de decisión

La estructura de decisión permite decidir entre ejecutar uno u otro conjunto de acciones en función de que se cumpla o no una determinada condición lógica.

En el capítulo anterior, explicamos el uso del `if`. Más adelante, en este mismo capítulo, veremos que existen otras estructuras selectivas como por ejemplo la decisión múltiple (`switch`) y el *if-inline*.

Comencemos por analizar un problema extremadamente simple.

Problema 2.1

Leer dos valores numéricos enteros e indicar cuál es el mayor y cuál es el menor. Considerar que ambos valores son diferentes.

Análisis

Los datos de entrada para este problema son los dos valores que ingresará el usuario. Nuestra tarea será compararlos para determinar cuál es mayor y cuál es menor.

El enunciado dice que debemos considerar que los valores serán diferentes. Por lo tanto, para nuestro análisis la posibilidad de que los valores sean iguales no será tomada en cuenta.

Llamaremos `a` al primer valor y `b` al segundo. Para compararlos utilizaremos una estructura de decisión que nos permitirá determinar si `a` es mayor que `b`. Si esto resulta verdadero, entonces `a` será el mayor y `b` será el menor. De lo contrario, si la condición anterior resulta falsa, `b` será el mayor y `a` el menor ya que, como mencionamos más arriba, `a` y `b` no serán iguales.

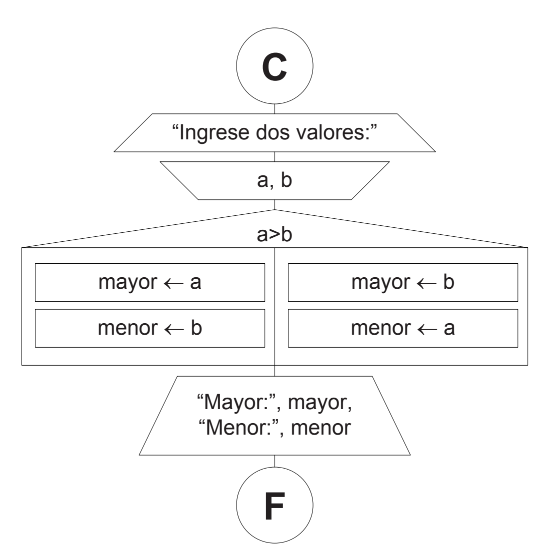


Fig. 2.1 Compara dos valores numéricos.

En el algoritmo utilizamos las variables `mayor` y `menor`. Una vez que comparamos `a` con `b` y determinamos cuál es el mayor y cuál es el menor, asignamos sus valores a estas variables y luego, mostramos sus contenidos.

El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   int a,b;
   int mayor,menor;

   printf("Ingrese dos valores: ");
   scanf("%d %d",&a,&b);

   if( a>b )
   {
      mayor=a;
      menor=b;
   }
   else
   {
      mayor=b;
      menor=a;
   }

   printf("Mayor: %d\n",mayor);
   printf("Menor: %d\n",menor);

   return 0;
}
```

■

Para resolver este problema, utilizamos una estructura de decisión que nos permitió determinar cuál es el mayor valor. Luego, por descarte, el otro es el menor.

2.3.1 Estructuras de decisión anidadas

Cuando en una estructura de decisión utilizamos otra estructura de decisión, decimos que ambas son estructuras anidadas. En el siguiente problema, utilizaremos estructuras de decisión anidadas para determinar, entre tres valores numéricos, cuál es el mayor, cuál es el del medio y cuál es el menor.

Problema 2.2

Leer tres valores numéricos enteros, indicar cuál es el mayor, cuál es el del medio y cuál, el menor. Considerar que los tres valores serán diferentes.

Veamos primero el algoritmo y luego lo analizaremos.

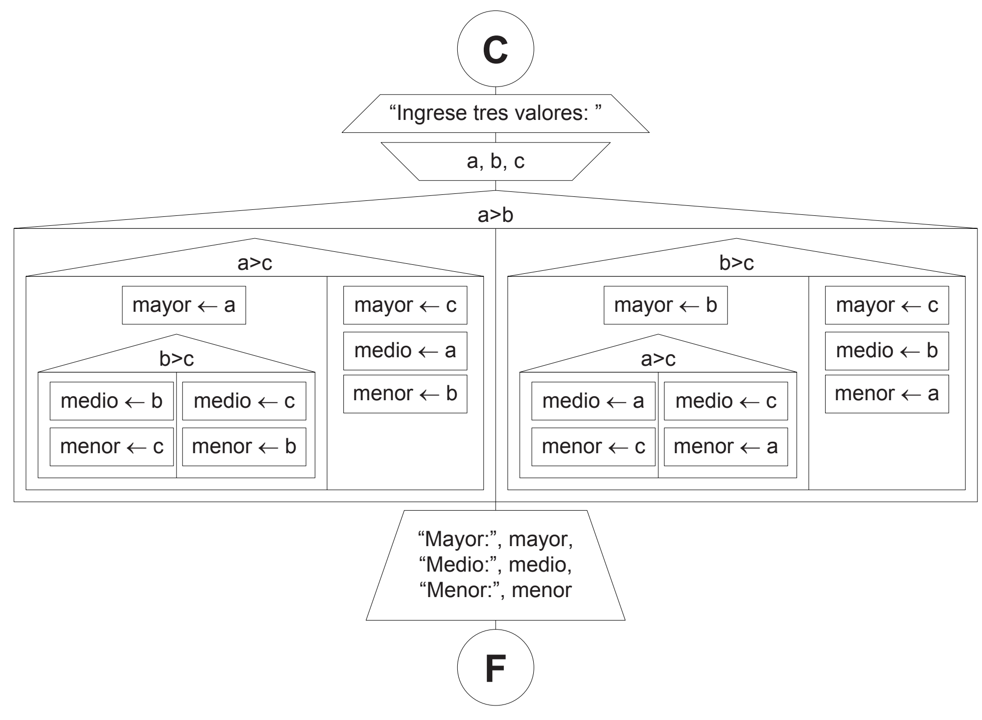


Fig. 2.2 Compara tres valores y determina cuál es mayor, medio y menor.

Análisis

Leemos los tres valores y comenzamos a comparar preguntando si `a>b`. Si esto se verifica entonces preguntamos si `a>c`. Si esto también se verifica entonces, como `a>b` y `a>c`, no hay dudas de que `a` es el mayor. Luego tenemos que comparar `b` y `c` para ver cuál está en segundo y en tercer lugar.

Si resulta que `a>b` pero no se verifica que `a>c` (es decir que `c` es mayor que `a`) será porque `c` es el mayor, `a` el medio y `b` el menor.

Por otro lado, si no se verifica que `a>b` preguntamos si `b>c`. Si esto es así, entonces el mayor será `b` (ya que `b` es mayor que `a` y `b` es mayor que `c`). Preguntamos si `a>c` y podremos deducir cuál está en segundo y tercer lugar.

Para finalizar, si es falso que `b>c` entonces el mayor será `c`, medio `b` y menor `a`.
El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   int a,b,c;
   int mayor,medio,menor;

   printf("Ingrese tres valores: ");
   scanf("%d %d %d",&a,&b,&c);

   if( a>b )
   {
      if( a>c )
      {
         mayor=a;
         if( b>c )
         {
            medio=b;
            menor=c;
         }
         else
         {
            medio=c;
            menor=b;
         }
      }
      else
      {
         mayor=c;
         medio=a;
         menor=b;
      }
   }
   else
   {
      if( b>c )
      {
         mayor=b;
         if( a>c )
         {
            medio=a;
            menor=c;
         }
         else
         {
            medio=c;
            menor=a;
         }
      }
      else
      {
```

```
         mayor=c;
         medio=b;
         menor=a;
      }
   }

   printf("Mayor: %d\n",mayor);
   printf("Medio: %d\n",medio);
   printf("Menor: %d\n",menor);

   return 0;
}
```

Para resolver este ejercicio recurrimos al uso de estructuras de decisión anidadas. El lector habrá notado que, a medida que anidamos más estructuras de decisión, resulta más complicado de seguir el código fuente.
Quizás un mejor análisis del problema hubiera sido el siguiente:
Si `a>b && a>c` entonces el mayor es `a`, el del medio será el máximo valor entre `b` y `c` y el menor será el mínimo valor entre estos.
Si la condición anterior no se verifica será porque `a` no es el mayor valor. Supongamos entonces que `b>a && b>c`. En este caso, el mayor será `b` y los valores medio y menor serán el máximo entre `a` y `c` y el mínimo entre `a` y `c` respectivamente.
Siguiendo este análisis, el algoritmo podría plantearse de la siguiente manera:

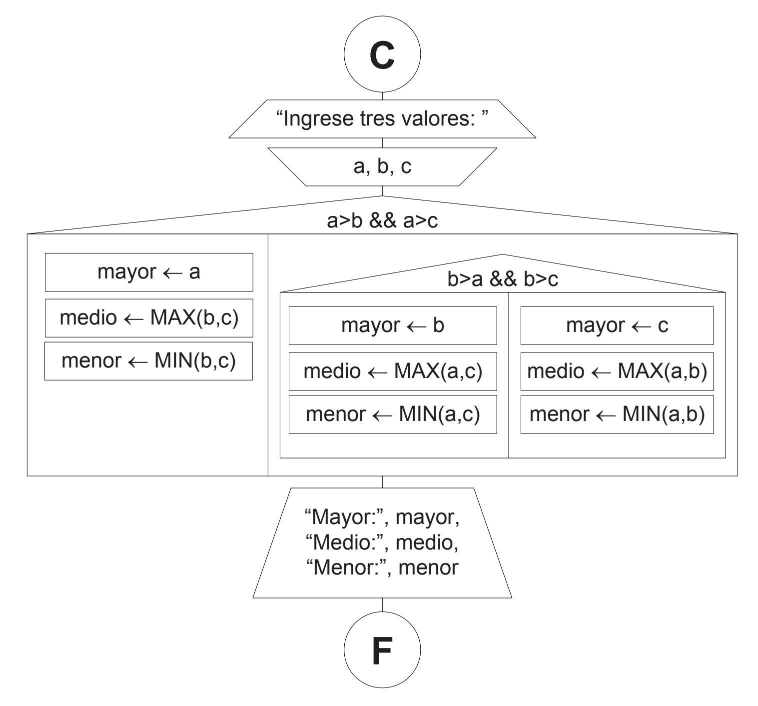


Fig. 2.3 Compara tres valores y determina cuál es mayor, medio y menor.

Para implementar esta solución, utilizaremos la estructura de selección en línea o *if-inline*.

2.3.2 Selección en línea o if-inline

Esta estructura implementa un `if` en una única línea de código y funciona así:

```
resultado = condicion ? expresion1:expresion2;
```

Si `condicion` resulta verdadera el *if-inline* retorna la expresión ubicada entre el signo de interrogación y los dos puntos. Si `condicion` resulta falsa entonces el valor de retorno será la expresión ubicada inmediatamente después de los dos puntos.

Por ejemplo: sean las variables `a` y `b` enteras y cada una con un valor numérico, y la variable `mayor` también entera, entonces:

```
mayor = a>b?a:b; // asigno a mayor el maximo valor entre a y b
```

Esta línea debe interpretarse de la siguiente manera: si se verifica que `a>b` entonces el *if-inline* retorna `a` (expresión ubicada entre el signo de interrogación y los dos puntos). Si la expresión lógica no se verifica entonces el valor de retorno del *if-inline* será `b` (expresión ubicada luego de los dos puntos). El resultado del *if-inline* lo asignamos a la variable `mayor`.

Utilizando el *if-inline* podemos codificar la segunda versión del problema 2.2 haciendo que sea más legible al reducir la cantidad de "ifes" anidados.

```
#include <stdio.h>

int main()
{
   int a,b,c;
   int mayor,medio,menor;

   printf("Ingrese tres valores: ");
   scanf("%d %d %d",&a,&b,&c);

   if( a>b && a>c )
   {
      mayor=a;
      medio=b>c?b:c; // el mayor entre b y c
      menor=b<c?b:c; // el menor entre b y c
   }
   else
   {
      if( b>a && b>c )
      {
         mayor=b;
         medio=a>c?a:c; // el mayor entre a y c
         menor=a<c?a:c; // el menor entre a y c
      }
      else
      {
         mayor=c;
         medio=a>b?a:b; // el mayor entre a y b
         menor=a<b?a:b; // el menor entre a y c
      }
   }
```

```
   printf("Mayor: %d\n",mayor);
   printf("Medio: %d\n",medio);
   printf("Menor: %d\n",menor);

   return 0;
}
```

2.3.3 Macros

Las macros son directivas de preprocesador con las que podemos relacionar un nombre con una expresión.

Por ejemplo:

```
#define MAX(x,y) x>y?x:y
#define MIN(x,y) x<y?x:y
```

El preprocesador de C reemplazará cada macro por la expresión que representa. Así, podemos invocar a la macro `MAX` de la siguiente manera:

```
mayor = MAX(a,b);
```

donde `a` y `b` son argumentos que le pasamos a la macro `MAX`. El preprocesador reemplazará la línea anterior por la siguiente línea:

```
mayor = a>b?a:b
```

Utilizando estas macros podemos mejorar aún más la legibilidad del código del problema 2.2.

```
#include <stdio.h>

// definimos las macros
#define MAX(x,y) x>y?x:y
#define MIN(x,y) x<y?x:y

int main()
{
   int a,b,c;
   int mayor,medio,menor;

   printf("Ingrese tres valores: ");
   scanf("%d %d %d",&a,&b,&c);

   if( a>b && a>c )
   {
      mayor=a;
      medio=MAX(b,c);
      menor=MIN(b,c);
   }
   else
   {
      if( b>a && b>c )
      {
         mayor=b;
         medio=MAX(a,c);
         menor=MIN(a,c);
```

```
      }
      else
      {
         mayor=c;
         medio=MAX(a,b);
         menor=MIN(a,b);
      }
   }

   printf("Mayor: %d\n",mayor);
   printf("Medio: %d\n",medio);
   printf("Menor: %d\n",menor);

   return 0;
}
```

■

2.3.4 Selección múltiple (switch)

La estructura de selección múltiple permite tomar una decisión en función de que el valor de una variable o el resultado de una expresión numérica entera coincidan o no con alguno de los valores indicados en diferentes casos o con ninguno de estos. Gráficamente, la representaremos así:

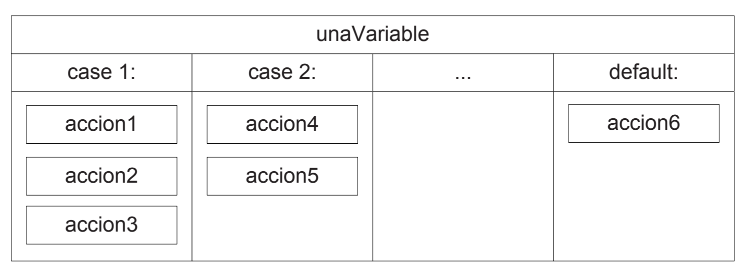


Fig. 2.4 Representación gráfica de la estructura de selección múltiple.

Este gráfico debe interpretarse de la siguiente manera: si el valor de `unaVariable` es 1 entonces se ejecutarán las acciones `accion1`, `accion2` y `accion3`. En cambio, si `unaVariable` vale 2 se ejecutarán las acciones `accion4` y `accion5`. Podemos agregar tantos casos como necesitemos. Por último, podemos indicar la opción `default` que representa a todos los otros casos que no fueron indicados explícitamente. A continuación, vemos la sintaxis genérica de esta estructura.

```
switch(expresion)
{
   case expresion_cte_1:
      sentencia_1;
      break;
   case expresion_cte_2:
      sentencia_2;
      break;
    :
   case expresion_cte_n:
      sentencia_n;
      break;
   [default:
      sentencia;]
}
```

Los casos deben representarse con valores numéricos literales o constantes. El caso `default` es opcional.

Problema 2.3

Leer un valor numérico que representa un día de la semana. Se pide mostrar por pantalla el nombre del día considerando que el lunes es el día 1, el martes es el día 2 y así, sucesivamente.

Análisis

Este problema se puede resolver fácilmente utilizando una estructura de selección múltiple como veremos a continuación:

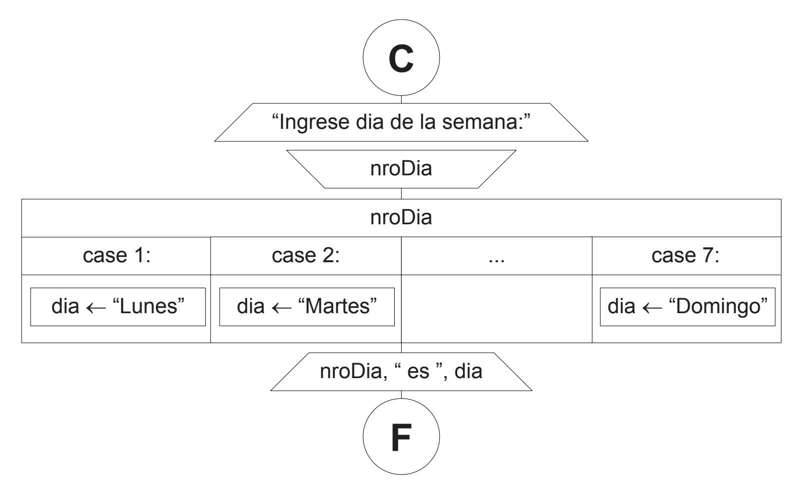


Fig. 2.5 Muestra el día de la semana según el número de día ingresado.

En el diagrama leemos el número de día en la variable `nroDia` y luego, utilizamos una estructura de selección múltiple (`switch`) con los casos 1, 2, 3, 4, 5, 6 y 7. Si el usuario ingresó el número de día 1, entonces la estructura ingresará por el caso `case 1` donde le asignamos la cadena "Lunes" a la variable `dia`. Análogamente, si el usuario ingresa el valor 2, entraremos por `case 2` y le asignaremos a `dia` la cadena "Martes" y así, sucesivamente.

La codificación es la siguiente:

```
#include <stdio.h>
#include <string.h>

int main()
{
   int nroDia;
   char dia[10];

   printf("Ingrese dia de la semana: ");
   scanf("%d",&nroDia);
```

```
   switch( nroDia )
   {
      case 1:
         strcpy(dia,"Lunes");  // asigno a dia la cadena "Lunes"
         break;
      case 2:
         strcpy(dia,"Martes"); // asigno a dia la cadena "Martes"
         break;
      case 3:
         strcpy(dia,"Miercoles");
         break;
      case 4:
         strcpy(dia,"Jueves");
         break;
      case 5:
         strcpy(dia,"Viernes");
         break;
      case 6:
         strcpy(dia,"Sabado");
         break;
      case 7:
         strcpy(dia,"Domingo");
         break;
   }

   printf("%d es %s\n",nroDia,dia);

   return 0;
}
```

Es muy importante poner la sentencia `break` al finalizar el conjunto de acciones que se ejecutan dentro de cada caso ya que si la omitimos se ejecutarán secuencialmente todas las acciones de todos los casos subsiguientes.

Es decir, supongamos que el usuario ingresa el día número 5 y omitimos poner los *breaks* entonces el programa ingresará por `case 5`, asignará “Viernes” a `dia`, luego le asignará “Sábado” y luego “Domingo”. Por lo tanto, la salida será:

```
5 es Domingo
```

Para asignar el nombre de cada día a la variable `dia`, no utilizamos el operador de asignación, lo hacemos a través de la función `strcpy`. Esto lo explicaremos a continuación.

2.3.5 Asignación de valores alfanuméricos (función `strcpy`)

Como comentamos anteriormente las cadenas de caracteres tienen un tratamiento especial ya que en C se implementan sobre *arrays* (conjuntos) de caracteres. Por este motivo, no podemos utilizar el operador de asignación `=` para asignarles valor. Tenemos que hacerlo a través de la función de biblioteca `strcpy` (definida en el archivo “string.h”) como vemos en las siguientes líneas de código:

```
// defino un "conjunto" de 10 variables de tipo char
char s[10];

// strcpy asigna cada uno de los caracteres de "Hola"
// a cada una de las variables del conjunto s
strcpy(s,"Hola");
```

Esta función toma cada uno de los caracteres de la cadena “Hola” y los asigna uno a uno a los elementos del conjunto `s`.

Gráficamente, podemos verlo así:

1. Definimos un *array* de 10 caracteres (o un conjunto de 10 variables de tipo `char`)
   `char s[10];`

   s = {

   }

2. Asignamos uno a uno los caracteres de la cadena “Hola” a los caracteres de `s`.
   `strcpy(s, "Hola");`

   s = {

'H'	'o'	'l'	'a'	**'\0'**					

   }

Como vemos `strcpy` asigna el *i-ésimo* carácter de la cadena “Hola” al *i-ésimo* carácter del conjunto o *array* `s`.

Además `strcpy` agrega el carácter especial `'\0'` (léase “barra cero”) que delimita el final de la cadena. Es decir que si bien `s` tiene espacio para almacenar 10 caracteres nosotros solo estamos utilizando 5. Cuatro para la cadena “Hola” más 1 para el `'\0'`.

Vale decir entonces que en un *array* de *n* caracteres podremos almacenar cadenas de, a lo sumo, *n*-1 caracteres ya que siempre se necesitará incluir el carácter de fin de cadena `'\0'` al final.

En otros lenguajes como Java o Pascal, las cadenas de caracteres tienen su propio tipo de datos, pero lamentablemente en C el manejo de cadenas es bastante más complicado. Por este motivo, lo estudiaremos en detalle más adelante, pero considero conveniente mencionar una cosa más: en el Capítulo 1, se expuso el siguiente ejemplo:

```
#include <stdio.h>

int main()
{
   char nombre[] = "Pablo";
   int edad = 39;
   double altura = 1.70;

   printf("Mi nombre es %s, tengo %d anios y mido %lf metros.\n"
          ,nombre
          ,edad
          ,altura);

   return 0;
}
```

Aquí utilizamos el operador de asignación `=` para asignar el valor “Pablo” a la variable `nombre`. Esto solo se puede hacer al momento de definir la variable. Incluso, como no hemos dimensionado la cantidad de caracteres que el *array* `nombre` puede contener, C dimensionará el conjunto de caracteres con tantos elementos como sea necesario para mantener la cadena “Pablo” más 1 para contener el `'\0'`.

Recomiendo al lector modificar el programa anterior de la siguiente manera:

```
#include <stdio.h>

int main()
{
   char nombre[20];

   nombre = "Pablo";

   // :
   // todo lo demas...
   // :

   return 0;
}
```

Aquí estamos intentando asignar “Pablo” a `nombre`, pero en una línea posterior a la declaración de la variable. Al compilar obtendremos el siguiente error:

```
datospersona.c: In function 'main':
datospersona.c:8: error: incompatible types in assignment
```

2.4 Estructura de repetición

La estructura de repetición, también llamada “estructura iterativa” o “ciclo de repetición”, permite ejecutar una y otra vez un conjunto de acciones en función de que se verifique una determinada condición lógica.

Según sea exacta o inexacta la cantidad de veces que el ciclo iterará podemos clasificar a la estructura de repetición de la siguiente manera:

- Estructura de repetición inexacta que itera entre 0 y *n* veces
- Estructura de repetición inexacta que itera entre 1 y *n* veces
- Estructura de repetición exacta que itera entre *i* y *n* veces, siendo $i <= n$

2.4.1 Estructuras de repetición inexactas

Llamamos así a las estructuras que iteran una cantidad variable de veces. En C estas estructuras son el ciclo `while` y el ciclo `do-while` y las representamos así:

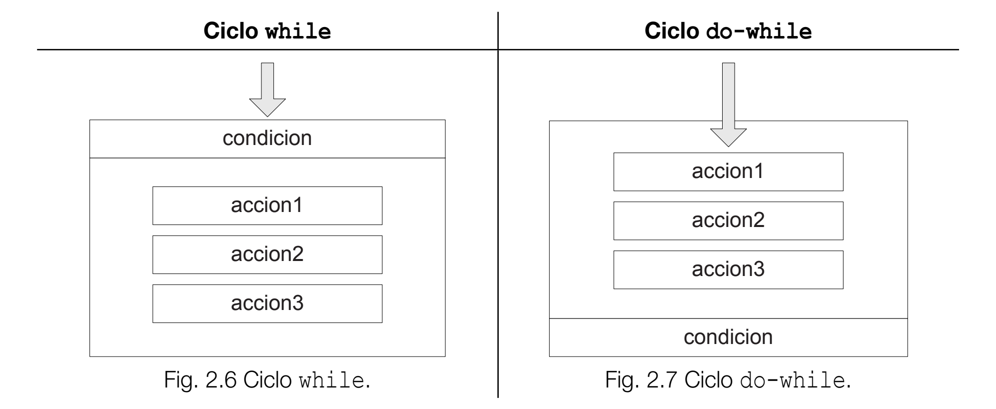


Fig. 2.6 Ciclo `while`.

Fig. 2.7 Ciclo `do-while`.

El ciclo `while` itera mientras que se cumpla la condición lógica indicada en su cabecera. Si al llegar a esta estructura la condición resulta ser falsa entonces el ciclo no iterará ni siquiera una vez. Por eso, decimos que se trata de un ciclo de repeticiones inexacto que puede iterar entre 0 y *n* veces. Además, como la condición se evalúa antes de ingresar al ciclo decimos que el `while` es un ciclo con "precondición".

En cambio, la entrada al ciclo `do-while` no está condicionada, por lo tanto, las acciones encerradas dentro de esta estructura se realizarán al menos una vez. Luego de esto se evaluará la condición lógica ubicada al pie de la estructura, que continuará iterando mientras que esta condición resulte verdadera. En este caso, decimos que se trata de un ciclo de repeticiones inexacto que iterará entre 1 y *n* veces. El `do-while` es un ciclo con "poscondición".

Problema 2.4

Se ingresa por teclado un conjunto de valores numéricos enteros positivos, se pide informar, por cada uno, si el valor ingresado es par o impar. Para indicar el final se ingresará un valor cero o negativo.

Análisis

Los datos de entrada de este problema son los números que ingresará el usuario. No sabemos cuántos números va a ingresar, solo sabemos que el ingreso de datos finalizará con la llegada de un valor cero o negativo. Por esto, mientras que el número ingresado sea mayor que cero tenemos que "procesarlo" para indicar si es par o impar.

Para resolver este problema, utilizaremos un ciclo de repetición que iterará mientras que el número ingresado sea mayor que cero. Luego, dentro de la estructura lo procesaremos para determinar e informar si el número es par o impar.

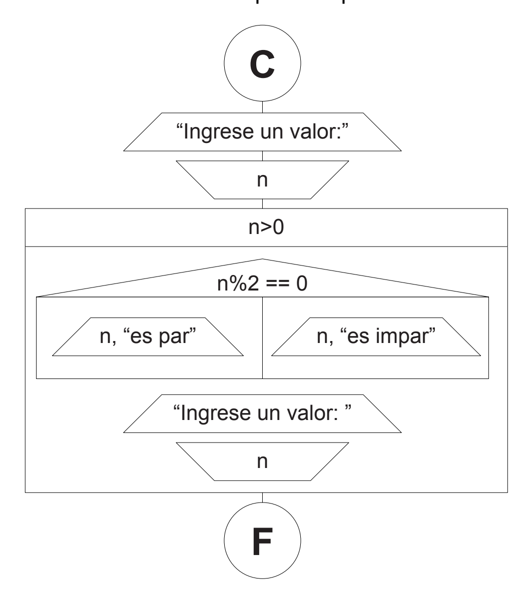


Fig. 2.8 Lee un conjunto de números e indica cuál es par y cuál es impar.

Como comentamos en el capítulo anterior, dentro del ciclo de repeticiones debe suceder "algo" que haga que, en algún momento, la condición lógica se deje de cumplir. En este caso, por cada iteración volvemos a leer en la variable `n` el siguiente número del conjunto. Por lo tanto, cuando el usuario ingrese un valor cero o negativo la condición lógica resultará falsa y el ciclo dejará de iterar.

Notemos que si el conjunto de datos está vacío el usuario solo ingresará un valor cero o negativo con el que la condición del ciclo resultará falsa y, directamente, no ingresará.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int n;

   printf("Ingrese un valor: ");
   scanf("%d",&n);

   while( n>0 )
   {
      if( n%2 == 0 )
      {
         printf("%d es par\n",n);
      }
      else
      {
         printf("%d es impar\n",n);
      }

      printf("Ingrese un valor: ");
      scanf("%d",&n);
   }

   return 0;
}
```

2.4.2 Estructuras de repetición exactas

A diferencia de las estructuras inexactas, las estructuras exactas permiten controlar la cantidad de veces que van a iterar. En general, se llaman "ciclos *for*" y definen una variable de control que toma valores sucesivos comenzando desde un valor inicial v_i y finalizando en un valor final v_n. Así, el `for` itera exactamente $v_n - v_i + 1$ veces y en cada iteración la variable de control tomará los valores v_i, v_i+1, v_i+2, ...,v_n.

En C el ciclo `for` se implementa como una mezcla entre el "*for* tradicional" y el ciclo `while`. Gráficamente, lo representaremos así:

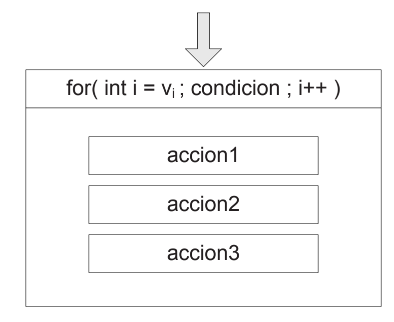


Fig. 2.9 Representación gráfica del ciclo `for`.

La cabecera del ciclo se compone de tres secciones separadas por `;` (punto y coma). En la primera sección, definimos la variable de control como una variable entera (en este caso es la variable `i`) y le asignamos un valor inicial. En la tercera sección, definimos la modalidad de incremento que se le aplicará a la variable de control. Con `i++` le indicamos al `for` que en cada iteración incremente en 1 el valor de `i`.

En la segunda sección, definimos una condición lógica. El for iterará incrementando el valor de la variable `i` de uno en uno y mientras que se verifique dicha condición. Es decir, si queremos que el ciclo itere exactamente *n* veces con una variable `i` variando entre 0 y *n*-1 tendremos que definir la cabecera del for de la siguiente manera:

```
for( int i=0; i<n; i++ )
```

Problema 2.5

Desarrollar un algoritmo que muestre por pantalla los primeros *n* números naturales considerando al 0 (cero) como primer número natural.

Análisis

El único dato de entrada que recibe el algoritmo es el valor *n* que el usuario deberá ingresar por teclado. Luego, la salida será: 0, 1, 2, ..., *n*-2, *n*-1. Es decir, si el usuario ingresa el valor 3 entonces la salida del algoritmo debe ser 0,1,2. Y si el usuario ingresa el valor 5 la salida será 0,1,2,3,4.

Para resolver el algoritmo utilizaremos una variable `i` cuyo valor inicial será 0. Luego mostramos su valor y lo incrementamos para que pase a ser 1. Si repetimos esta operación mientras que `i` sea menor que *n* tendremos resuelto el problema.

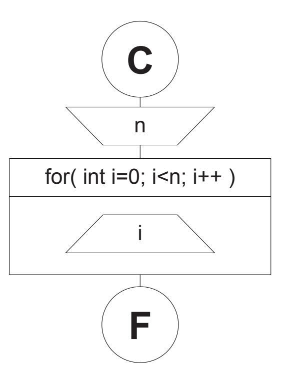


Fig. 2.10 Muestra los primeros *n* números naturales incluyendo el 0.

Para simplificar la lectura de los diagramas, en algunos casos comenzaré a omitir los mensajes con indicaciones para el usuario. Por ejemplo, aquí decidí no incluir el mensaje “Ingrese un valor numérico”. Sin embargo, en la codificación sí los incluiré.

Como vemos, el `for` prácticamente resuelve todo el problema ya este ciclo define la variable `i`, le asigna el valor inicial 0, la incrementa de uno en uno e itera mientras que su valor sea menor que `n`. Dentro del `for` todo lo que queda por hacer es mostrar el valor de la variable `i`.

Veamos la codificación del algoritmo:

```c
#include <stdio.h>

int main()
{
   int n;

   printf("Ingrese un valor numerico: ");
   scanf("%d",&n);

   for(int i=0; i<n; i++)
   {
      printf("%d\n",i);
   }

   return 0;
}
```

Este problema también podemos resolverlo utilizando un ciclo `while` pero, entonces, la responsabilidad de incrementar la variable será nuestra. Para esto, en cada iteración luego de mostrar el valor de `i` incrementaremos su valor asignándole su valor actual "más 1". En este caso, decimos que `i` es un contador.

2.4.3 Contadores

Llamamos "contador" a una variable cuyo valor iremos incrementando manualmente, de uno en uno, dentro de un ciclo de repetición. Para incrementar el valor de una variable numérica, tenemos que asignarle "su valor actual más 1" de la siguiente manera:

```c
// incrementamos el valor de la variable x
x = x+1;
```

La línea anterior debe leerse así: "a `x` le asigno `x` más 1". Si el valor actual de `x` es 2, luego de incrementarlo su valor será 3. Los contadores siempre deben tener un valor inicial.

Resolveremos el problema anterior reemplazando el ciclo `for` por un ciclo `while` e incrementando manualmente un contador.

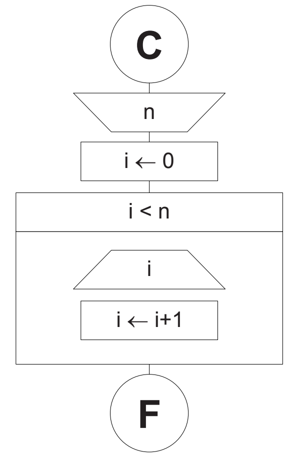


Fig. 2.11 Muestra los primeros *n* números naturales utilizando un contador.

Es muy importante asignarle un valor inicial a la variable que implementa el contador.

En el diagrama le asignamos el valor inicial 0 a la variable `i`. De no haberlo hecho entonces la condición del `while` no tendría sentido. Luego, al mostrar `i` estaríamos mostrando algo incierto. Tampoco podríamos asignarle a `i` "su valor actual más 1" porque `i` no tendría ningún valor actual.

Llamamos "inicializar" a la acción de asignarle valor inicial a una variable. Una variable que no está inicializada no tiene asignado ningún valor concreto. Es decir, no tiene valor.

Volviendo al diagrama inicializamos la variable `i`, pero no inicializamos la variable `n`. Esto se debe a que `n` tomará su valor inicial luego de que el usuario lo ingrese por teclado.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int i, n;

   printf("Ingrese un valor numerico: ");
   scanf("%d",&n);

   i = 0;
   while( i<n )
   {
      printf("%d\n",i);
      i = i+1;
   }

   return 0;
}
```

2.4.4 Acumuladores

Llamamos así a una variable cuyo valor iremos incrementando en cantidades variables dentro de un ciclo de repetición. Esto lo logramos de la siguiente manera:

```
x = x+n;
```

La diferencia entre un acumulador y un contador es que el contador se incrementa de a 1 en cambio el acumulador se incrementa de a *n* siendo *n* una variable. Obviamente, un contador es un caso particular de acumulador.

Problema 2.6

Determinar la sumatoria de los elementos de un conjunto de valores numéricos. Los números se ingresarán por teclado. Se ingresará un cero para finalizar.

Análisis

Para resolver este problema, utilizaremos un ciclo `while` dentro del cual acumularemos en la variable `suma` cada uno de los valores que ingrese el usuario. El ciclo de repetición iterará mientras que el valor ingresado sea distinto de cero.

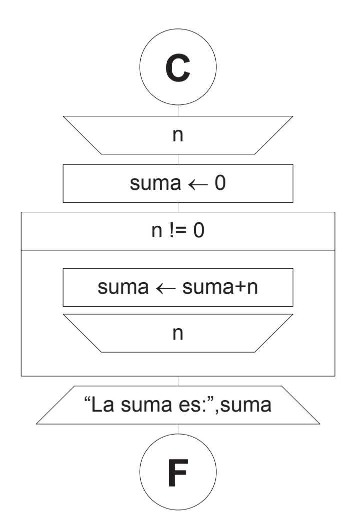


Fig. 2.12 Suma un conjunto de valores numéricos.

La variable `suma` es un acumulador porque dentro del ciclo `while` su valor se incrementa de a `n` unidades siendo `n` un valor potencialmente distinto en cada iteración ya que toma cada uno de los valores que ingresa el usuario.

2.4.5 Seguimiento del algoritmo y “prueba de escritorio”

La prueba de escritorio es una herramienta que nos permite comprobar si el algoritmo realmente funciona como esperamos. Básicamente, consiste en definir un conjunto de datos arbitrarios que llamaremos “lote de datos” y utilizarlos en un seguimiento, paso a paso, de cada una de las acciones del algoritmo controlando los valores que irán tomando las variables, las expresiones lógicas y la salida que se va a emitir.

Analizaremos una prueba de escritorio para el diagrama del problema anterior, considerando el siguiente lote de datos {5, 2, 3, 0}.

Comenzamos leyendo `n` que, según nuestro lote de datos, tomará el valor 5. Inicializamos `suma` en 0 y luego entramos a un ciclo de repetición para iterar mientras que `n` sea distinto de 0 o, dicho de otro modo, mientras que la expresión “`n` es distinto de cero” sea verdadera. Como `n` vale 5 se verifica la condición lógica e ingresamos al `while`. Dentro del `while` asignamos a `suma` su valor actual (cero) más el valor de `n` (cinco) por lo que ahora `suma` vale 5. Luego volvemos a leer `n` que tomará el valor 2 y volvemos a la cabecera del ciclo de repetición para evaluar si corresponde iterar una vez más. Como `n` (que vale 2) es distinto de cero volvemos a ingresar al `while`, asignamos a `suma` su valor actual (cinco) más el valor de `n` (dos) por lo que `suma` ahora vale 7. Luego leemos `n` que tomará el valor 3. La condición del ciclo se sigue verificando porque `n` (que vale 3) es distinto de cero. Entonces asignamos a `suma` su valor actual (que es 7) más el valor de `n` (que es 3). Ahora `suma` vale 10. Volvemos a leer `n` que tomará el valor cero y evaluamos la condición del ciclo. Como la expresión “`n` es distinto de cero” resulta falsa el ciclo no volverá a iterar, salimos del `while` y mostramos el valor de la variable `suma`: 10.

Todo este análisis puede resumirse en la siguiente tabla que iremos llenando paso a paso, siguiendo el algoritmo y considerando que ingresan uno a uno los valores del lote de datos.

n	suma	n!=0	Acción	suma = suma+n	n	Salida
5	0					
		true	entro al while	5	2	
		true	entro al while	7	3	
		true	entro al while	10	0	
		false	salgo del while			La suma es: 10

La tabla debe leerse de arriba hacia abajo y de izquierda a derecha. El lector puede acceder al videotutorial que muestra el desarrollo de esta misma prueba de escritorio.

Por último, veamos la codificación del algoritmo:

```
#include <stdio.h>

int main()
{
   int n,suma;

   printf("Ingrese un valor numerico: ");
   scanf("%d",&n);

   suma=0;
   while( n!=0 )
   {
      suma = suma+n;

      printf("Ingrese el siguiente valor: ");
      scanf("%d",&n);
   }

   printf("La suma es: %d\n",suma);

   return 0;
}
```



Uso del *debugger* para depurar un programa

2.4.6 El debugger, la herramienta de depuración

El *debugger* es una herramienta que permite seguir paso a paso la ejecución del código fuente del programa. También permite monitorear los valores que toman las variables y evaluar las expresiones.

Las IDE integran el *debugger* con el editor de código fuente de forma tal que, dentro del mismo editor, el programador pueda seguir línea por línea la ejecución del programa y así, depurarlo de errores de lógica.

En los videos tutoriales se explica como *debuggear* (depurar) un programa utilizando el *debugger* de Eclipse.

Continuemos con más ejemplos.

Problema 2.7

Dado un conjunto de valores numéricos que se ingresan por teclado determinar el valor promedio. El fin de datos se indicará ingresando un valor igual a cero.

Análisis

Sabemos que para obtener el promedio de un conjunto de números tenemos que sumarlos y luego dividir la suma por la cantidad de elementos del conjunto. Por esto necesitaremos un acumulador para obtener la suma (lo llamaremos `suma`) y un contador para saber cuántos números fueron sumados (lo llamaremos `cant`). El resultado estará disponible luego de que hayamos procesado todos los elementos del conjunto. Esto es: fuera del ciclo de repetición.

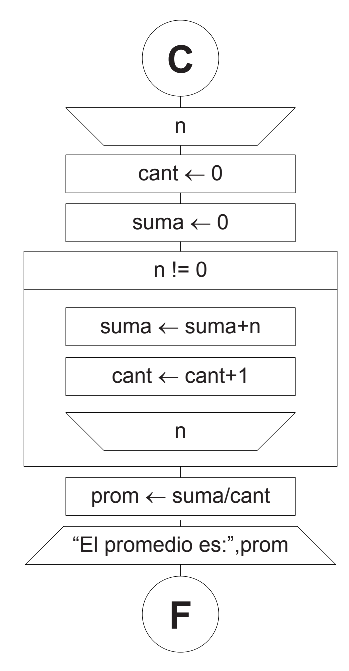


Fig. 2.13 Obtiene el promedio de un conjunto de valores numéricos.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int n,cant,suma;
   double prom;

   printf("Ingrese un valor numerico: ");
   scanf("%d",&n);

   cant=0;
   suma=0;
```

```
   while( n!=0 )
   {
      suma = suma+n;
      cant = cant+1;

      printf("Ingrese el siguiente valor: ");
      scanf("%d",&n);
   }

   prom = (double)suma/cant;

   printf("El promedio es: %lf\n",prom);

   return 0;
}
```

Problema 2.8

Se ingresa un valor numérico por consola, determinar e informar si se trata de un número primo o no.

Análisis

Los números primos son aquellos que solo son divisibles por sí mismos y por la unidad. Es decir que un número `n` es primo si para todo entero `i` tal que `i>1 && i<n` se verifica que `n%i` es distinto de cero.

La estrategia que utilizaremos para resolver este problema será la siguiente: consideraremos que el número `n` que ingresa el usuario es primo y luego vamos a ver si entre 2 y `n-1` existe algún número `i` tal que `i` sea divisor de `n`. Si existe algún valor de `i` que haga que `n%i` sea cero será porque `i` es divisor de `n` y, por lo tanto, `n` no será primo.

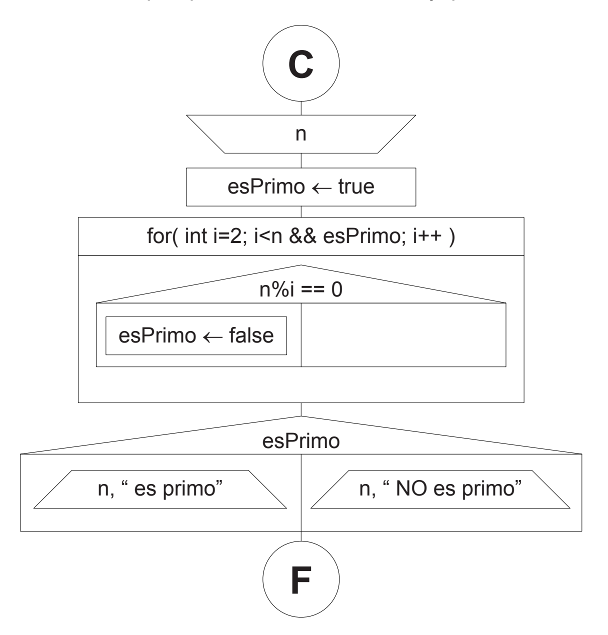


Fig. 2.14 Determina si un número es primo o no.

```
#include <stdio.h>

int main()
{
   int n,esPrimo;

   printf("Ingrese un valor numerico: ");
   scanf("%d",&n);

   esPrimo=1;

   for( int i=2; i<n && esPrimo; i++ )
   {
      if( n%i==0 )
      {
         esPrimo=0;
      }
   }

   if( esPrimo )
   {
      printf("%d es primo\n",n);
   }
   else
   {
      printf("%d NO es primo\n",n);
   }

   return 0;
}
```
■

2.4.7 Estructuras de repetición anidadas

Cuando una estructura de repetición encierra a otra estructura de repetición decimos que son estructuras anidadas.

Problema 2.9

Desarrollar un algoritmo que muestre los primeros *n* números primos siendo *n* un valor que debe ingresar el usuario.

Análisis

En este algoritmo el usuario ingresa la cantidad `n` de números primos que quiere ver. Por ejemplo, si `n` es 5 entonces el programa debe mostrar los primeros cinco números primos, es decir, 1, 2, 3, 5, 7.

La estrategia aquí será mantener dos contadores que llamaremos `num` y `cont`. A `num` lo vamos a inicializar en 1 y lo utilizaremos para evaluar si su valor actual es primo, luego lo incrementaremos. Cada vez que determinemos que `num` es primo tenemos que mostrarlo e incrementar a `cont.` Por lo tanto, `cont` contará la cantidad de números primos que hemos mostrado hasta el momento.

Para implementar esta estrategia, utilizaremos dos ciclos de repetición anidados. El ciclo interno iterará entre 2 y `num-1` para determinar si el valor de `num` es primo o no utilizando el mismo algoritmo del problema anterior. El ciclo externo iterará mientras que `cont` sea menor que `n`. Este ciclo controlará que mostremos exactamente los primeros `n` números primos.

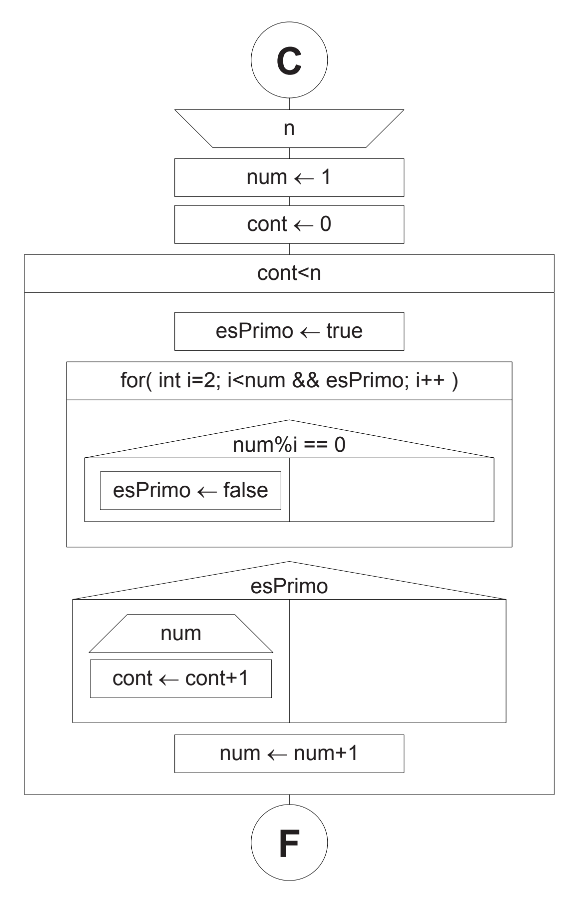


Fig. 2.15 Muestra los primeros n números primos.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int n,esPrimo;
   int num,cont;

   printf("Cuantos primos quiere ver? ");
   scanf("%d",&n);


   num=1;
   cont=0;

   while( cont<n )
   {
      esPrimo=1;
```

```
      for( int i=2; i<num && esPrimo; i++ )
      {
         if( num%i==0 )
         {
            esPrimo=0;
         }
      }

      if( esPrimo )
      {
         printf("%d\n",num);
         cont=cont+1;
      }

      num = num+1;
   }

   return 0;
}
```

2.4.8 Manejo de valores booleanos

En el Capítulo 1, hablamos sobre los tipos de datos y comentamos que existe un tipo de datos capaz de contener los valores lógicos “verdadero” y “falso” o, en inglés, *true* y *false*. A este tipo de datos lo llamamos “tipo booleano” o simplemente *boolean*.

En C no existe el tipo *boolean* como tal. En su lugar se usan las variables de tipo `int` y se considera que su valor de verdad es *false* si contienen 0 y *true* si contienen cualquier otro valor numérico.

Esto nos permite asignar en una variable de tipo `int` el resultado de una expresión lógica como veremos a continuación:

```
#include <stdio.h>

int main()
{
   int n, esPar;

   scanf("%d",&n);

   // asigno a esPar el resultado de la expresion n%2==0
   esPar = n%2==0;

   // si esPar es verdadero...
   if( esPar )
   {
      printf("%d es par\n",n);
   }

   return 0;
}
```

En este ejemplo asignamos a la variable `esPar` el resultado de la expresión lógica `n%2==0`. Luego preguntamos directamente por el valor de verdad de `esPar`.

Quizás pueda resultar confuso que un mismo tipo de datos se utilice para contener valores de naturaleza diferente. La siguiente tabla muestra como a una variable de tipo `int` se la puede tratar como tipo *boolean* y como tipo entero.

Tratamiento booleano	Tratamiento numérico

`// :`
`if( esPar )`
`{`
`   printf("%d es par\n",n);`
`}`
	`// :`
	`if( esPar!=0 )`
	`{`
	`   printf("%d es par\n",n);`
	`}`

Ambos tratamientos son correctos. Sin embargo, el más apropiado es el que se muestra en el cuadro de la izquierda porque estamos haciendo referencia al valor lógico de la variable `esPar`, no a su valor numérico.

2.4.9 Máximos y mínimos

En ocasiones necesitaremos encontrar el máximo o el mínimo valor dentro de un conjunto de valores numéricos. Los siguientes problemas nos ayudarán a estudiar este tema.

Problema 2.10

Dado un conjunto de valores numéricos indicar cuál es el mayor. El ingreso de datos finaliza con la llegada de un cero.

Análisis

Tenemos que leer los valores que el usuario ingresará por teclado y determinar cuál de estos es el mayor. Para encontrar un algoritmo que nos permita resolver el problema, primero pensemos como lo haríamos mentalmente.

Supongamos que trabajamos con el siguiente lote de datos {2, 6, 1, 7, 3, 0} y solo podemos acceder a un valor a la vez. Entonces leeremos un valor y lo consideraremos como "el mayor" hasta tanto no leamos otro que lo supere.

Acción	Valor que ingresa	Mayor hasta el momento
leo un valor	2	2
leo el siguiente valor	6	6
leo el siguiente valor	1	6
leo el siguiente valor	7	7
leo el siguiente valor	3	7
leo el siguiente valor	0	**7**

Como vemos, al finalizar el ingreso de datos podemos determinar que el mayor valor del conjunto, según nuestro lote, es el número 7.

Este algoritmo lo representamos de la siguiente manera:

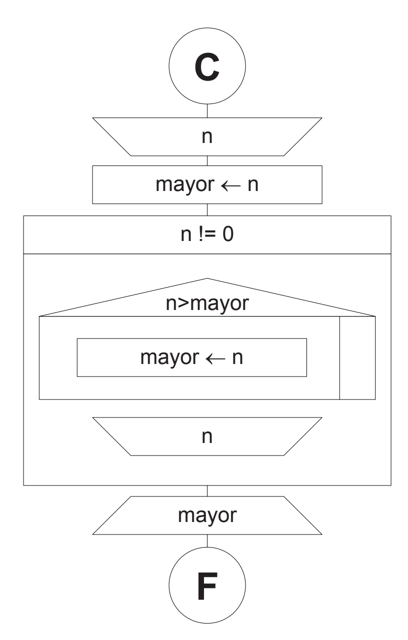


Fig. 2.16 Obtiene el mayor valor de un conjunto.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int n,mayor;

   printf("Ingrese un valor: ");
   scanf("%d",&n);

   mayor = n;

   while( n!=0 )
   {
      if( n>mayor )
      {
         mayor = n;
      }

      printf("Ingrese el siguiente valor: ");
      scanf("%d",&n);
   }

   printf("EL mayor es: %d\n", mayor);

   return 0;
}
```

■

Problema 2.11

Determinar el menor valor de un conjunto de números e indicar también su posición relativa dentro del mismo. El ingreso de datos finaliza con la llegada de un cero.

Análisis

El algoritmo para obtener el menor valor de un conjunto es análogo al que utilizamos para obtener el mayor. Leeremos un valor y este será el menor mientras no leamos otro que resulte ser más pequeño.

En este problema, además tenemos que indicar la posición relativa. Para analizar esto utilizaremos el mismo lote de datos del problema anterior: {2, 6, 1, 7, 3, 0}.

Según el lote de datos, el menor valor del conjunto es 1 y su posición relativa es 3 ya que se encuentra ubicado en el tercer lugar. Recordemos que el cero no debe ser tenido en cuenta porque su función es indicar el fin del ingreso de datos.

Para identificar la posición relativa del menor valor del conjunto, tendremos que usar un contador y guardar su valor actual cada vez que encontremos un número más chico que el que, hasta el momento, consideramos como el menor.

Acción	**Valor que ingresa**	**`posRel`**	**`menor`**	**`posRelMenor`**
Leo un valor	2	1	2	1
Leo el siguiente	6	2	2	1
Leo el siguiente	1	3	1	3
Leo el siguiente	7	4	1	3
Leo el siguiente	3	5	1	3
Leo el siguiente	0		**1**	**3**

El algoritmo es el siguiente:

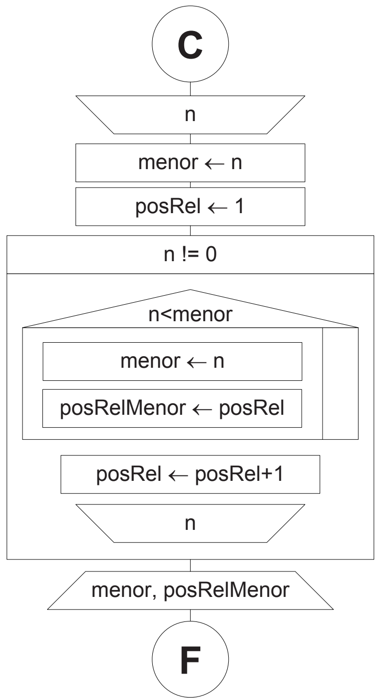


Fig. 2.17 Dado un conjunto de valores obtiene el menor y su posición relativa.

```
#include <stdio.h>

int main()
{
   int n,menor;
   int posRel,posRelMenor;

   printf("Ingrese un valor: ",n);
   scanf("%d",&n);

   menor = n;
   posRel = 1;

   while( n!=0 )
   {
      if( n<menor )
      {
         menor = n;
         posRelMenor = posRel;
      }


      posRel = posRel+1;

      printf("Ingrese un valor: ",n);
      scanf("%d",&n);
   }

   printf("Menor valor: %d, posicion relativa: %d\n"
         ,menor
         ,posRelMenor);

   return 0;
}
```

2.5 Contextualización del problema

Hasta aquí analizamos problemas que simplemente hablaban de conjuntos de valores numéricos. El objetivo ahora es trabajar con enunciados que hagan referencia a contextos reales como pueden ser sueldos, personas, calificaciones en exámenes, fechas, etcétera.

Problema 2.12

Se tiene una tabla o planilla con los resultados de la última llamada a examen de una materia, con la siguiente información:

- matrícula (valor numérico entero de 8 dígitos)
- nota (valor numérico entero de 2 dígitos entre 1 y 10)
- nombre (valor alfanumérico de 10 caracteres)

Se pide informar:

- Cantidad de alumnos que se presentaron a rendir examen
- Nota promedio
- Nombre y nota del alumno que obtuvo el mejor resultado (será único)

Para indicar el fin del ingreso de datos el operador ingresará un registro nulo con matrícula=0, nota=0 y nombre = “”.

Análisis

Comenzaremos por analizar un lote de datos con el formato de la tabla que describe el enunciado.

Matrícula	Nota	Nombre
15323423	5	Juan
13654234	2	Pedro
16319672	8	Marcos
12742411	6	Roque
0	0	

Recordemos que el lote de datos se genera con valores arbitrarios, por lo tanto, se trata solo de una suposición. Pero si este fuera el caso entonces los resultados que pide el enunciado del problema serían los siguientes:

1. Cantidad de alumnos que se presentaron a rendir examen: **4**
2. Nota promedio: (5+2+8+6) / 4 = **5.25**
3. Nombre y nota del alumno que obtuvo el mejor resultado (será único). “**Marcos**” y la nota obtenida es: **8**

Llamamos “registro” a cada fila de la tabla y “campo” a cada columna. En esta planilla, cada registro representa la nota que obtuvo un alumno y tiene los campos: matrícula, nota y nombre.

Utilizaremos un ciclo de repetición para leer uno a uno los registros de la tabla. Si dentro de este ciclo incrementamos un contador (que llamaremos `cantPres`) podremos determinar la cantidad de alumnos que se presentaron a rendir el examen ya que la tabla tiene tantos registros como alumnos rindieron la materia. Es decir, el contador se incrementará tantas veces como exámenes figuren en la tabla.

Para obtener la nota promedio, necesitaremos conocer la cantidad de presentaciones que hubo y la sumatoria de las notas obtenidas en dichas presentaciones. El primer dato lo tenemos en el contador `cantPres` que analizamos en el párrafo anterior. El segundo dato lo podemos obtener fácilmente acumulando cada nota en un acumulador que llamaremos `sumNotas`.

Al finalizar el ingreso de datos, esto es una vez terminado el ciclo de repetición, tendremos la cantidad de presentaciones a examen `cantPres` y podremos obtener el promedio como el cociente: `sumNotas/cantPres`.

Por último, tenemos que obtener la nota más alta para informar cuál fue el alumno que obtuvo el mejor resultado. Veamos el diagrama del algoritmo y luego analizaremos este punto.

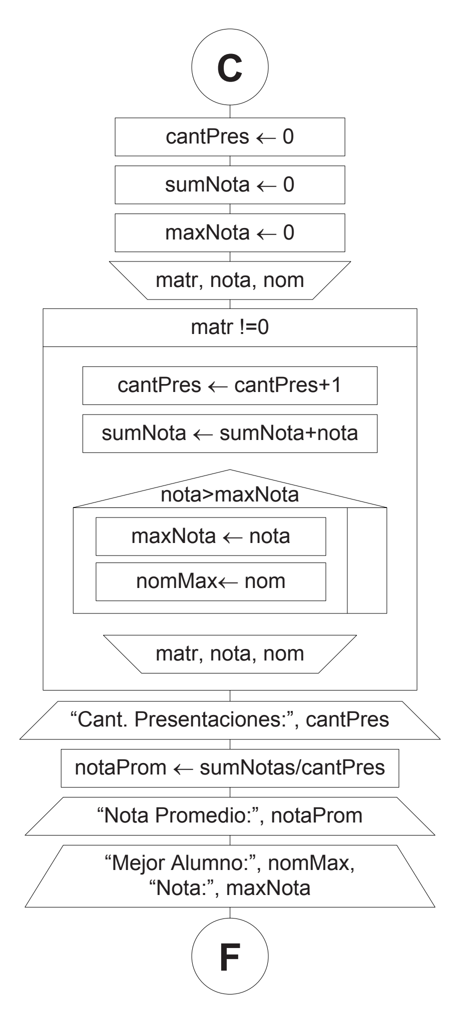


Fig. 2.18 Procesa presentaciones a examen y muestra datos estadísticos.

Para determinar el alumno de mejor rendimiento, utilizamos la variable `maxNota` a la que le dimos el valor inicial 0. Como las diferentes notas pueden variar entre 1 y 10 (según dice el enunciado) resulta que cero es una cota inferior de este conjunto. Es decir que, cualquiera sea la nota que figure en el primer registro de la planilla seguro será mayor que 0; por lo tanto, podemos asegurar que será mayor que `maxNota` con lo cual el algoritmo ingresará al `if`. Allí actualizamos el valor de `notaMax` y asignamos a `maxNom` el nombre del alumno cuya nota (hasta el momento) se considera la mejor.

Veamos el código:

```
#include <stdio.h>
#include <string.h>

int main()
{
   int cantPres,sumNota,maxNota;
   int matr,nota;
   char nom[11],nomMax[11];

   // inicializo las variables
   cantPres = 0;
   sumNota = 0;
   maxNota = 0;

   printf("Ingrese matricula, nota, nombre: ");
   scanf("%d %d %s",&matr,&nota,nom);

   while( matr!=0 )
   {
      cantPres = cantPres+1;
      sumNota = sumNota+nota;

      if( nota>maxNota )
      {
         maxNota = nota;
         strcpy(nomMax,nom);
      }

      printf("Ingrese matricula, nota, nombre: ");
      scanf("%d %d %s",&matr,&nota,nom);
   }

   // cantidad de presentaciones
   printf("Cantidad de presentaciones: %d\n",cantPres);

   // nota promedio
   double prom = (double)sumNota/cantPres;
   printf("Nota promedio: %lf\n",prom);

   // mejor alumno
   printf("Alumno de mejor rendimiento %s, nota: %d\n"
         ,nomMax
         ,maxNota);

   return 0;
}
```

Si ejecutamos el programa e ingresamos los datos del lote de datos que definimos más arriba obtendremos los siguientes resultados:

```
Ingrese matricula, nota, nombre: 15323423 5 Juan
Ingrese matricula, nota, nombre: 13654234 2 Pedro
Ingrese matricula, nota, nombre: 16319672 8 Marcos
Ingrese matricula, nota, nombre: 12742411 6 Roque
Ingrese matricula, nota, nombre: 0 0 0
Cantidad de presentaciones: 4
Nota promedio: 5.25
Alumno de mejor rendimiento Marcos, nota: 8
```

Problema 2.13

Se ingresa por consola un número entero que representa un sueldo que se debe pagar. Considerando que existen billetes de las denominaciones que se indican más abajo; informar, que cantidad de billetes de cada denominación se deberá utilizar, dando prioridad a los de valor nominal más alto.

Denominaciones ($) = {100, 50, 20, 10, 5, 2, 1}

Análisis

Comencemos por identificar los datos de entrada; en este caso, será el sueldo que se debe abonar. También son conocidas las denominaciones de los billetes que se utilizarán para pagarlo.

Ahora bien, supongamos que el sueldo a pagar es de: $5217 (pesos cinco mil doscientos diecisiete). Fácilmente, con un cálculo mental, podemos deducir que para abonar dicha cifra necesitaremos disponer de las siguientes cantidades de billetes:

Cantidad	Denominación
52	100
0	50
0	20
1	10
1	5
1	2
0	1

Intuitivamente, llegamos a este resultado dividiendo el importe del sueldo por el billete de mayor denominación. Esto es: $5217/$100 = 52; es decir, usaremos 52 billetes de $100 para pagar $5200 y pagaremos el residuo de $17 con billetes de menor denominación.

Obviamente, no podemos pagar $17 con billetes de $50 ni de $20; sin embargo, podemos utilizar un billete de $10. Con esto, solo quedará cubrir el residuo de $7 que pagaremos utilizando un billete de $5 y uno de $2.

De este análisis se desprende el siguiente algoritmo que resuelve el problema.

Si llamamos *v* al sueldo que vamos a abonar y *d* al billete de mayor denominación entonces:

1. Dividir el sueldo *v* por el valor nominal del billete *d*. El cociente de esta división indicará cuantos billetes de dicha denominación se requiere utilizar.
2. Repetir el paso 1, pero considerando que *v* es el residuo originado en la división anterior y que *d* será, ahora, el segundo billete mayor denominación ($50). Así hasta haber pasado los billetes de todas las denominaciones disponibles.

```
#include <stdio.h>

int main()
{
   int v;

   // ingreso el sueldo a pagar
   printf("Ingrese el valor a pagar: $");
   scanf("%d",&v);

   int denom = 100;
   int cant = v/denom;
   int residuo = v%denom;
   printf("%d billetes de $%d\n",cant,denom);

   // repito la operacion considerando solo el residuo
   v = residuo;
   denom = 50;
   cant = v/denom;
   residuo = v%denom;
   printf("%d billetes de $%d\n",cant,denom);

   // repito la operacion considerando solo el residuo
   v = residuo;
   denom = 20;
   cant = v/denom;
   residuo = v%denom;
   printf("%d billetes de $%d\n",cant,denom);

   // repito la operacion considerando solo el residuo
   v = residuo;
   denom = 10;
   cant = v/denom;
   residuo = v%denom;
   printf("%d billetes de $%d\n",cant,denom);

   // repito la operacion considerando solo el residuo
   v = residuo;
   denom = 5;
   cant = v/denom;
   residuo = v%denom;
   printf("%d billetes de $%d\n",cant,denom);

   // repito la operacion considerando solo el residuo
   v = residuo;
   denom = 2;
   cant = v/denom;
   residuo = v%denom;
   printf("%d billetes de $%d\n",cant,denom);

   // repito la operacion considerando solo el residuo
   v = residuo;
   denom = 1;
   cant = v/denom;
   residuo = v%denom;
   printf("%d billetes de $%d\n",cant,denom);

   return 0;
}
```

Como podemos ver, el siguiente bloque de código:

```
v = ?;
denom = ?;
cant = v/denom;
residuo = v%denom;
printf("%d billetes de $%d\n",cant,denom);
```

se repite tantas veces como cantidad de elementos tenga el conjunto de denominaciones de los billetes disponibles.

Vemos también que entre repetición y repetición, solo cambian los valores de `v` (sueldo) y `denom` (denominación del billete).

En el siguiente capítulo, veremos que, utilizando funciones, podremos mejorar dramáticamente la legibilidad de esta solución.

2.6 Resumen

En este capítulo incrementamos el nivel de complejidad de los problemas y, en consecuencia, el de los algoritmos que los resuelven. Obviamente, el nivel sigue siendo introductorio, pero este incremento nos permitió conocer algunos recursos que son fundamentales en la lógica algorítmica: los contadores, los acumuladores y la mecánica para determinar el máximo y el mínimo valor dentro de un conjunto.

También vimos cómo desarrollar una prueba de escritorio y, con la ayuda de un videotutorial, cómo utilizar el *debugger*.

En el siguiente capítulo, estudiaremos la metodología *top-down* que resultará clave para el desarrollo y buen diseño de nuestros algoritmos.

2.7 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

2.7.1 Mapa conceptual

2.7.2 Autoevaluaciones

2.7.3 Videotutorial

2.7.3.1 Uso del debugger para depurar un programa

2.7.4 Presentaciones*

3

Funciones, modularización y metodología top-down

Contenido

Objetivos del capítulo

- Comprender la necesidad y la importancia de lograr abstracción.
- Simplificar problemas resolviéndolos progresivamente mediante el "refinamiento sucesivo" de su análisis e implementación; metodología *top-down*.
- Conocer y utilizar funciones de la biblioteca estándar de C.
- Desarrollar nuestras propias funciones.
- Utilizar funciones y macros para que el código fuente de nuestros programas resulte claro y legible.
- Entender la diferencia entre argumentos y parámetros, valor y referencia.
- Determinar el alcance (*scope*) de las variables; locales, globales, automáticas.
- Usar variables estáticas para mantener oculta la implementación de una función.

Competencias específicas

- Conocer las características principales del lenguaje C.
- Construir programas utilizando estructuras condicionales y repetitivas para aumentar su funcionalidad.

3.1 Introducción

En el Capítulo 1, explicamos que los algoritmos están presentes en todas nuestras acciones. Aplicamos algoritmos para resolver situaciones cotidianas tales como cruzar una calle, preparar una taza de té o leer un libro.

Si prestamos atención a los pasos que seguimos cuando, por ejemplo, preparamos una taza de té naturalmente llegaremos a la siguiente enumeración:

1. Ir a la cocina
2. Calentar agua
3. Preparar la taza
4. Poner el agua en la taza

Sabemos también que "ir a la cocina" implica "caminar", y "caminar" implica "dar un paso tras otro" y "dar un paso" implica "levantar, adelantar y bajar un pie", "levantar el pie" implica "tensionar un conjunto de músculos", etcétera.

Pensemos ahora este mismo algoritmo al revés, analizando desde la acción más particular hasta llegar a la acción más general. Tendríamos que "tensionar los músculos" para "levantar un pie" para "dar un paso" para "ir a la cocina". Sería verdaderamente enloquecedor e impracticable.

3.2 Conceptos iniciales

3.2.1 Metodología top-down

La metodología *top-down* propone pensar la solución de un problema en base a una secuencia de acciones de nivel general cuyos pasos iremos refinando sucesivamente. Esto significa partir de un análisis global y refinarlo hasta llegar a un análisis particular.

Aplicando esta metodología a la resolución de problemas computacionales podremos resolver situaciones complejas con una simpleza y naturalidad verdaderamente sorprendente.

3.2.2 Módulos o subprogramas

Llamamos "módulo", "rutina" o "subprograma" a cada una de las acciones que ejecutamos en nuestro algoritmo y que luego vamos a desarrollar.

En el ejemplo de preparar una taza de té, invocamos a los módulos "caminar hacia la cocina", "calentar el agua", etc. Luego tendremos que desarrollar cada uno de estos módulos para los cuales, probablemente, necesitemos invocar a otros módulos que también tendremos que desarrollar.

3.2.3 Funciones

Cuando aplicamos algoritmos a la resolución de problemas computacionales los módulos se implementan como funciones.

Una función es un subprograma invocado desde el programa principal (o desde otra función) que ejecuta una tarea determinada y luego, retorna el control al programa o función que la invocó.

3.2.4 Funciones de biblioteca

Los lenguajes de programación proveen bibliotecas de funciones. Llamamos "biblioteca de funciones" a un conjunto de funciones que resuelven tareas estándar que, generalmente, son requeridas en la mayoría de los programas.

Ya utilizamos las funciones `printf` y `scanf` que permiten mostrar y leer datos por consola. También usamos la función `strcpy` para asignar una cadena de caracteres en un `char[]` (léase "char array").

En C las funciones de biblioteca están definidas en archivos con extensión ".h" (inicial de *header*).

Veamos algunos de los archivos `.h` estándar del lenguaje de programación C.

Archivo	Descripción	Algunas de las funciones que define
`stdio.h`	Entrada y salida estándar	`printf`, `scanf`, `sprintf`, `getc`, `putc`, `fopen`, `fclose`, `fscanf`, etc.
`string.h`	Utilidades para manejo de cadenas de caracteres	`strcpy`, `strcat`, `ctrcmp`, etc.
`math.h`	Funciones matemáticas	`sin`, `cos`, `log`, `abs`, `pow`, etc.
`stdlib.h`	Funciones de memoria y utilitarias	`atoi`, `itoa`, `rand`, `malloc`, `free`, etc.

3.2.5 Invocación a funciones de biblioteca

Para invocar en nuestro programa a una función de biblioteca tenemos que incluir el archivo `.h` que contiene su definición. Esto lo hacemos con la directiva `#include` como vemos a continuación:

```
#include <stdio.h>

int main()
{
   printf("Hola Mundo !!!\n");

   return 0;
}
```

En el ejemplo invocamos a la función `printf` cuya definición se encuentra en el archivo `stdio.h`. Por este motivo, al inicio de nuestro programa, utilizamos la directiva `#include` para incluirlo.

3.3 Funciones definidas por el programador

La modularización es la base de la metodología *top-down* y, como comentamos más arriba, en C los módulos se implementan como funciones.

Las funciones realizan una determinada tarea "en función" de un conjunto de argumentos que reciben como parámetros y luego retornan un resultado. A este resultado lo llamaremos "valor de retorno". El valor que retorna una función, generalmente, dependerá de los valores de los argumentos que le pasemos.

Una función se compone de una cabecera y un cuerpo. En la cabecera se especifican los parámetros con los que la función va a trabajar y el tipo de dato del valor de retorno. El cuerpo es la codificación del algoritmo de la función.

3.3.1 Prototipo de una función

El prototipo, también llamado *header*, describe la lista de argumentos que la función espera recibir y el tipo de dato de su valor de retorno.

Por ejemplo, si pensamos en una función que retorne el valor absoluto de un número, su prototipo será el siguiente:

```
double valorAbsoluto(double);
```

Esta línea describe una función llamada `valorAbsoluto` que recibe un único argumento de tipo `double` y retorna un valor de este mismo tipo de datos.

O bien, podríamos definir una función para unificar tres valores numéricos enteros que representen el día, mes y año de una fecha en un único valor numérico entero con formato *aaaammdd*.

```
long unificarFecha(int, int, int);
```

Este prototipo describe a la función `unificarFecha` que recibe tres argumentos de tipo `int` y retorna un valor de tipo `long`.

3.3.2 Invocar a una función

Si desde el programa principal queremos invocar a una función previamente tendremos que prototiparla. Esto no es más que agregar una línea de código con el prototipo de la función que vamos a utilizar.

En el siguiente programa, le pedimos al usuario que ingrese un valor numérico, luego invocamos a la función `valorAbsoluto` para obtener el valor absoluto del número que el usuario ingresó y mostrarlo por pantalla.

En el código podemos ver que el prototipo de la función `valorAbsuluto` precede a la codificación del programa principal.

```
#include <stdio.h>

// prototipo de la funcion
double valorAbsoluto(double);

int main()
{
   double v, a;
   printf("Ingrese un valor numerico: ");
   scanf("%lf",&v);

   // invoco a la funcion
   a = valorAbsoluto(v);

   printf("El valor absoluto de %lf es %lf\n",v,a);

   return 0;
}
```

Obviamente, para desarrollar una función no alcanza con definir el prototipo. También la tendremos que programar.

3.3.3 Desarrollo de una función

Para desarrollar una función, tenemos que definir su cabecera y su cuerpo. La cabecera es idéntica al prototipo, pero le pone nombre a los argumentos que, desde el punto de vista de la función llamaremos "parámetros".

Veamos el código de la función `valorAbsoluto` que retorna el valor absoluto de un número que recibe como parámetro.

```
double valorAbsoluto(double d)
{
   double ret = d;

   if( ret<0 )
   {
      ret = -ret;
   }

   return ret;
}
```

Otra forma de codificarla podría ser utilizando un *if-inline*:

```
double valorAbsoluto(double d)
{
   return d<0?-d:d;
}
```

Veamos ahora el programa completo que le muestra al usuario el valor absoluto del número que ingresa por teclado.

```
#include<stdio.h>

// prototipo de la funcion
double valorAbsoluto(double);

// programa principal
int main()
{
   double v, a;
   printf("Ingrese un valor numerico: ");
   scanf("%lf",&v);

   // invoco a la funcion
   a = valorAbsoluto(v);

   printf("El valor absoluto de %lf es %lf\n",v,a);

   return 0;
}

// desarrollo de la funcion
double valorAbsoluto(double d)
{
   return d<0?-d:d;
}
```

3.3.4 Convención de nomenclatura para funciones

Si bien los nombres que podemos darle a las funciones tienen las mismas restricciones que aplican sobre los nombres de las variables, en este libro, adoptaremos la siguiente convención de nomenclatura:

- Nombres simples: deben escribirse completamente en minúscula. Por ejemplo `sumar`, `procesar`, `abrir`, `cerrar`, etcétera.
- Nombres compuestos: si el nombre de la función está compuesto por dos o más palabras entonces cada una, exceptuando la primera, debe comenzar en mayúscula. Por ejemplo: `valorAbsoluto`, `obtenerFechaNacimiento`, `procesarValores`, etcétera.

3.3.5 Funciones que no retornan ningún valor (tipo de datos `void`)

En C existe el tipo de datos `void` (nulo). Este tipo se utiliza, entre otras cosas, para indicar que una función no retornará ningún valor. Por ejemplo, a continuación veremos la función `saludar` que simplemente muestra por pantalla un cordial saludo.

```
void saludar()
{
   printf("Hola !!!\n");
   return;
}
```

Como la función `saludar` es `void`, la sentencia `return` queda vacía y no retornan ningún valor. Incluso, esta sentencia podría no estar. Es opcional.

3.3.6 Archivos de cabecera (.h)

Los archivos de cabecera (o *header*) se utilizan para contener prototipos de funciones, macros, constantes y (como veremos más adelante) tipos de datos definidos por el programador. Luego, estas líneas de código serán incluidas en el programa principal mediante la directiva `include`.



Mantener archivos separados para las funciones y el programa principal.

3.3.7 Archivos de funciones (.c)

En general, no es una buena idea desarrollar las funciones en el mismo archivo que el programa principal porque esto impide que puedan invocarse desde otro programa.

Lo aconsejable es que las funciones estén desarrolladas en otros archivos de código fuente de forma tal que cualquier programa que las necesite las pueda utilizar.

Siguiendo estas pautas, el ejemplo anterior podría reestructurarse en los siguientes tres archivos:

- `funciones.h`
- `funciones.c`
- `principal.c`

Veamos el contenido de cada uno de estos archivos:

`funciones.h`

```
// prototipos de funciones
double valorAbsoluto(double);
```

funciones.c

```
// desarrollo de las funciones
double valorAbsoluto(double d)
{
   return d<0?-d:d;
}
```

principal.c

```
#include <stdio.h>
#include "funciones.h"

// programa principal
int main()
{
   double v, a;
   printf("Ingrese un valor numerico: ");
   scanf("%lf",&v);

   // invoco a la funcion
   a = valorAbsoluto(v);

   printf("El valor absoluto de %lf es %lf\n",v,a);

   return 0;
}
```

Notemos que para incluir el contenido del archivo `funciones.h` utilizamos "comillas", en cambio para incluir el archivo `stdio.h` utilizamos los signos "menor y mayor".

Se utilizan "comillas" cuando el archivo que vamos a incluir está ubicado en la misma carpeta que el programa que lo incluye. Se utilizan los signos "menor y mayor" cuando el archivo que vamos a incluir está ubicado en otra carpeta. En este caso, la carpeta debe ser referenciada por la variable de entorno `INCLUDE`.

3.4 Legibilidad y reusabilidad del código

Las funciones proveen dos ventajas insustituibles: hacen que el algoritmo, y consecuentemente su codificación, sea mucho más claro y simple de seguir (legibilidad) y, por otro lado, permiten que un mismo conjunto de acciones pueda ser utilizado en diferentes programas (reusabilidad).

Para ejemplificar esto resolveremos el siguiente problema.

Problema 3.1

Leer seis valores numéricos enteros. Los primeros 3 representan el día, el mes y el año de una fecha, los tres restantes representan los mismos atributos de otra. Se pide determinar e informar cuál de las dos fechas es posterior.

Análisis

Los datos de entrada en este problema son los seis valores enteros que ingresará el usuario. Los llamaremos `d1`, `m1`, `a1` y `d2`, `m2`, `a2`. Los primeros tres son los atributos de una fecha que llamaremos `f1`. Los tres restantes representan a otra fecha que llamaremos `f2`.

La fecha posterior entre `f1` y `f2` será aquella que tenga un valor mayor en su atributo año. Si ambas tienen el mismo valor en este atributo entonces la fecha posterior será la que tenga el mayor mes. Si el mes también coincide entonces la fecha posterior será aquella que tenga el mayor día.

A simple vista, este problema puede resolverse anidando estructuras de decisión. Sin embargo, podemos plantear un enfoque diferente si pensamos en representar a cada fecha como un número entero de 8 dígitos donde los primeros 4 representan al año, los siguientes 2 representan al mes y los dos últimos dígitos representan al día.

Con esta representación numérica, el orden cronológico de las fechas coincidirá con su orden cardinal por lo que la fecha posterior será aquella que numéricamente resulte mayor.

Vamos a verificarlo. Entre las fechas 25/12/1981 y 14/11/2009 es evidente que la posterior es 14/11/2009. Veamos sus representaciones numéricas: El número 19811225 es menor que 20091114, por lo tanto, esta última es la fecha posterior.

Si pudiéramos obtener fácilmente las representaciones numéricas de `f1` y `f2` en función de sus atributos el programa principal se limitaría a informar cuál de estas es la mayor. Es decir, sería extremadamente simple de resolver.

3.4.1 Abstracción

Para resolver fácilmente el programa principal, desarrollaremos la función `unificarFecha` cuyo prototipo será el siguiente:

```
long unificarFecha(int,int,int);
```

Esta función espera recibir el día, mes y año de una fecha y retorna su representación numérica con formato *aaaammdd*.

Si aceptamos la idea de que contamos con la función `unificarFecha` (aunque todavía no la hayamos desarrollado) podremos abstraernos del problema y concentrarnos en el desarrollo del programa principal.

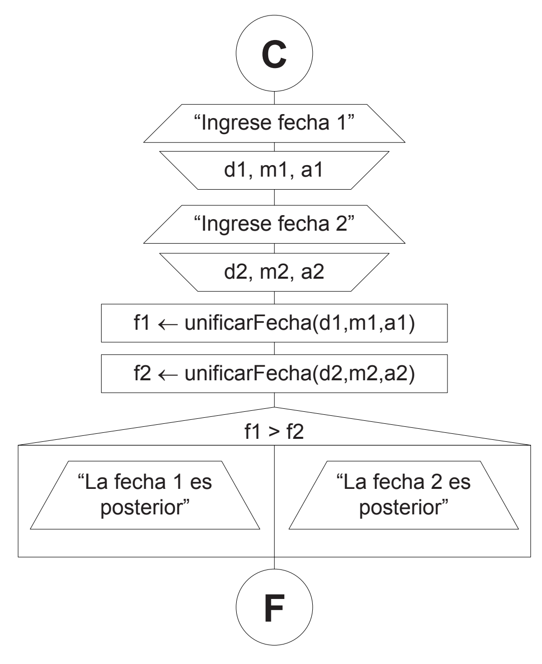


Fig. 3.1 Compara dos fechas e informa cuál es posterior.

Ahora podemos concentrarnos en el problema particular de unificar los atributos día, mes y año de una fecha en un único valor numérico entero con formato *aaaammdd*.

Este problema ya lo hemos analizado en los capítulos anteriores. El valor numérico que debe retornar la función se obtiene multiplicando el año*10000, luego sumando el mes*100 y luego el día.

Por ejemplo si el día es 5, el mes es 12 y el año es 2003 entonces:

año*10000 + mes*100 + día =

2003*10000 + 12*100 + 5 = **20031205**

La representación gráfica de la función `unificarFecha` es la siguiente:

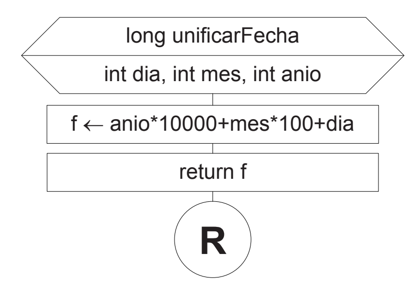


Fig. 3.2 Unifica los atributos de una fecha.

Veamos ahora la codificación del algoritmo:

fechas.h

```
long unificarFecha(int,int,int);
```

fechas.c

```
long unificarFecha(int dia, int mes, int anio)
{
   int f = anio*10000+mes*100*dia;
   return f;
}
```

principal.c

```
#include <stdio.h>
#include "fechas.h"

int main()
{
   int d1,m1,a1;
   int d2,m2,a2;
   long f1,f2;

   printf("Ingrese fecha 1: ");
   scanf("%d %d %d",&d1,&m1,&a1);

   printf("Ingrese fecha 2: ");
   scanf("%d %d %d",&d2,&m2,&a2);
```

```
    // invoco a la funcion para unificar f1 y luego f2
    f1 = unificarFecha(d1,m1,a1);
    f2 = unificarFecha(d2,m2,a2);


    if( f1>f2 )
    {
       printf("La fecha 1 es posterior\n");
    }
    else
    {
       printf("La fecha 2 es posterior\n");
    }

    return 0;
}
```

3.4.2 Argumentos y parámetros

Muchas veces los términos “argumento” y “parámetro” se utilizan como sinónimos pero, en realidad, no lo son. Existe una diferencia sutil entre ellos:

- A la función “le pasamos” argumentos.
- La función “recibe parámetros”.

Es decir, desde el punto de vista del programa en el cual invocamos a la función le pasamos argumentos, pero, desde el punto de vista de la función que es invocada, esta recibe parámetros.

En el siguiente gráfico, podemos ver el prototipo de la función `unificarFecha`, luego una invocación en la que le pasamos los argumentos `d1`, `m1` y `a1` y, por último, su cabecera. Podemos observar que cada argumento que le pasamos en la invocación se corresponde con un parámetro de la función, según su posición.

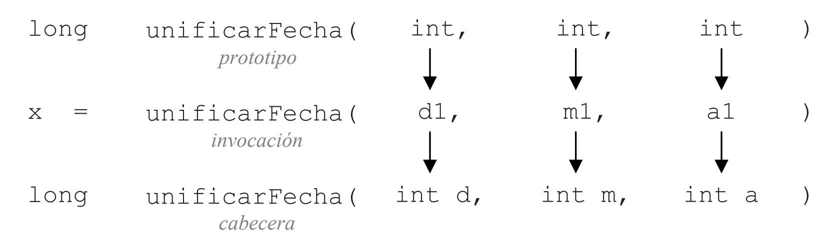


Fig. 3.3 Prototipo, invocación y cabecera de una función.

Dentro de la función, el parámetro `d` toma el valor del argumento que le pasamos en primer lugar que, en este caso, es `d1`. El parámetro `m` toma el valor del segundo argumento y el parámetro `a` toma el valor del tercer y último argumento.

Los nombres de los parámetros no tienen porque coincidir con los nombres de los argumentos. Lo que importa es la posición.

Problema 3.2

Mostrar los primeros *n* números primos siendo *n* un valor que ingresará el usuario.

Análisis

Este problema ya fue analizado y resuelto en el Capítulo 2. Sin embargo, veremos que si desarrollamos una función que indique si un número entero es primo o no, el programa será mucho más fácil de resolver y el algoritmo resultará más claro y legible.

Supongamos que contamos con la función `esPrimo` con el siguiente prototipo:

```
int esPrimo(int);
```

La función recibe un valor entero y retorna *true* o *false* según dicho valor corresponda o no un número primo.

Contando con la función `esPrimo` el problema se resuelve con dos contadores: `i` y `cont`. Por cada valor de `i`, verificamos si corresponde a un número primo y, en ese caso, mostramos su valor e incrementaremos a `cont`. Cuando `cont` sea mayor o igual que `n`, habremos mostrado los primeros `n` números primos.

Veamos el diagrama.

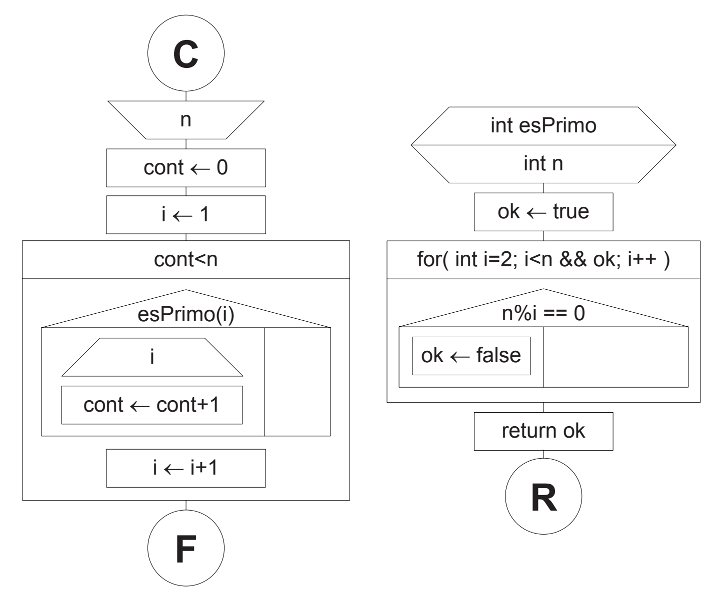


Fig. 3.4 Muestra los primeros números primos usando una función.

Desde el punto de vista del programa principal, disponemos de la función `esPrimo` que nos permite determinar si un número es primo o no. Con esta facilidad, el algoritmo se resuelve "probando" con cada uno de los números naturales para ver cuáles son primos e incrementando un contador por cada número primo que encontramos.

Desde el punto de vista de la función, el problema es mucho más acotado: simplemente tenemos que determinar si el parámetro `n` es o no un número primo. Para esto, verificaremos si `n` tiene algún divisor entre 2 y `n-1`. Es decir, si existe un valor `i` que esté entre 2 y `n-1` tal que `n%i` sea cero entonces `i` será divisor y el resultado ya no será `ok`, es decir que `n` no será número primo. Veamos ahora la codificación:

numeros.h

```c
int esPrimo(int);
```

numeros.c

```c
int esPrimo(int n)
{
   int ok=1;
   for( int i=2; i<n && ok; i++ )
   {
      if( n%i==0 )
      {
         ok = 0;
      }
   }

   return ok;
}
```

principal.c

```c
#include <stdio.h>
#include "numeros.h"

int main()
{
   int n, cont, i;

   printf("Cuantos primos quiere ver? ");
   scanf("%d",&n);

   cont=0;
   i=1;

   while( cont<n )
   {
      if( esPrimo(i) )
      {
         printf("%d\n",i);
         cont = cont+1;
      }

      i = i+1;
   }

   return 0;
}
```

3.5 Alcance de las variables (scope)

3.5.1 Variables locales

Las variables que definimos dentro de una función existen únicamente dentro de la función y cuando esta finaliza las variables se destruyen. Decimos que son “variables locales a la función” o simplemente “variables locales”.

Si desde una función invocamos a otra función, esta última definirá sus propias variables locales que nada tendrán que ver con las variables locales de la función que la invocó. Tal es así que, aunque ambas funciones definan variables con los mismos nombres, cada una de estas variables estará representando una porción de memoria diferente.

En el ejemplo anterior, en el programa principal (función `main`) definimos las variables `i` y `n` (entre otras) e invocamos a la función `esPrimo`. En esta función definimos la variable `i` (dentro del `for`) y utilizamos `n` como identificador de su único parámetro.

Las variables locales `i` y `n` que definimos y utilizamos dentro de la función `esPrimo` no tienen nada que ver con las variables locales `i` y `n` que definimos y utilizamos en la función `main`.

3.5.2 Variables globales

Las variables que definimos fuera del cuerpo de una función pueden ser vistas y manipuladas por todas las funciones que se encuentren desarrolladas más abajo.

Veamos el siguiente ejemplo:

```
#include <stdio.h>

void incrementar();

// i es una variable global
int i=0;

int main()
{
   // puedo utilizar i porque es global
   while(i<10)
   {
      printf("%d\n",i);
      incrementar();
   }

   return 0;
}

void incrementar()
{
   // puedo utilizar i porque es global
   i = i+1;
}
```

En este ejemplo definimos la variable global `i` inicializada en cero. Luego en la función `main` mostramos su valor e invocamos a la función `incrementar` para incrementarlo. Ambas funciones tienen acceso y pueden manipular la misma variable.

El uso de variables globales es una muy mala práctica de programación que solo aporta problemas de mantenibilidad y legibilidad del código.

En este libro no utilizaremos variables globales, por lo que este apartado fue incluido solo a título informativo.

3.6 Argumentos por valor y referencia

Cuando le pasamos argumentos a una función, esta solo recibe su valor. La función puede utilizar el valor del parámetro, pero no puede modificar el contenido de la variable original.

En el siguiente ejemplo, la función `permutar` intenta permutar los valores de los argumentos que recibe como parámetros. Veremos que aunque la función modifica sus valores luego, al retornar al programa principal, las variables que le pasamos a la función mantienen sus valores originales.

```
#include <stdio.h>

// prototipo de la funcion permutar
void permutar(int,int)

// programa principal
int main()
{
   int a=5,b=10;

   // muestro los valores de a y b
   printf("en main: a=%d, b=%d\n",a,b);

   // invoco a la funcion para permutar los valores de a y b
   permutar(a,b);

   // muestro los valores de a y b
   printf("en main: a=%d, b=%d\n",a,b);

   return 0;
}

void permutar(int x,int y)
{
   int aux;

   // muestro los valores de x e y
   printf("...en permutar: x=%d, y=%d\n",x,y);

   // permuto los valores de los argumentos
   aux = x;
   x = y;
   y = aux;

   // muestro los valores de x e y
   printf("...en permutar: x=%d, y=%d\n",x,y);
}
```

La salida de este programa será la siguiente:

```
en main: a=5, b=10
...en permutar: x=5, y=10
...en permutar: x=10, y=5
en main: a=5, b=10
```

En `main` las variables `a` y `b` tienen los valores 5 y 10 respectivamente. Luego invocamos a `permutar` que recibe ambos valores en los parámetros `x` e `y`. Cuando, dentro de esta función, intercambiamos los valores de `x` e `y` el cambio resulta efectivo, pero al retornar al programa principal las variables `a` y `b` mantienen su valor original.

En el lenguaje de programación C, los argumentos siempre se pasan por valor. Si queremos que una función pueda modificar el contenido de un argumento entonces tenemos que pasarle es su dirección de memoria. En este caso, estaremos pasando una referencia.

Antes de analizar como pasar argumentos por referencia considero conveniente estudiar con cierto nivel de detalle el uso de punteros y direcciones de memoria.

3.6.1 Punteros y direcciones de memoria

En el Capítulo 1, hablamos del operador de dirección `&` (léase "ampersand") y explicamos que al anteponer este operador al identificador de una variable hacemos referencia a su dirección de memoria. Es decir que, si `a` es una variable entonces `&a` es la dirección de memoria donde se aloja el valor que contiene la variable `a`.

Las direcciones de memoria son valores de un tipo de datos particular llamado "puntero".

Si la variable `a` es de tipo `int` entonces el tipo de dato de su dirección de memoria es "puntero a `int`" o simplemente `int*` (léase "int asterisco"). Veamos el siguiente ejemplo:

```
#include <stdio.h>

int main()
{
   // defino la variable a
   int a = 0;

   // defino la variable p de tipo "puntero a int"
   int* p;

   // a p le asigno la direccion de memoria de a
   p = &a;

   // al contenido de p le asigno el valor 12
   *p = 12;

   // muestro el valor de la variable a
   printf("a=%d\n",a);

   return 0;
}
```

La salida de este programa será:

```
a = 12
```

Aquí definimos la variable `a` de tipo `int` y la variable `p` de tipo `int*`. Como `p` tiene la capacidad de contener una dirección de memoria le asignamos la dirección de la variable `a` que obtenemos anteponiendo a esta variable el operador de dirección `&`.

Luego accedemos al espacio de memoria direccionado por `p` a través de `*p` (léase "asterisco p") y asignamos allí el valor 12. Como este espacio de memoria corresponde a la variable `a`, indirectamente, estaremos asignando el valor 12 a esta variable.

En C el asterisco se utiliza para definir variables de tipo puntero, pero también se utiliza para hacer referencia al contenido alojado en el espacio de memoria direccionado por un puntero.

3.6.2 El operador de indirección `*` (asterisco)

Este es el operador inverso de `&`. Aplicando el operador `&` a una variable obtenemos su dirección de memoria. En cambio, aplicando `*` a un puntero obtenemos el contenido que se alojado en esa dirección. Veamos el siguiente ejemplo:

```
#include <stdio.h>

int main()
{
   int a = 0;

   *(&a) = 10;

   printf("a=%d\n",a);

   return 0;
}
```

Aquí accedemos al contenido ubicado en la dirección de memoria de la variable `a` para asignar el valor 10. Esto es equivalente a asignarle el valor directamente a la variable `a`. La salida del programa será:

`a=10.`

En C este tema es fundamental ya que, al no existir el paso de argumentos por referencia, la única manera que tenemos para hacer que una función pueda modificar el contenido de los argumentos que recibe es trabajar directamente con sus direcciones.

Le recomiendo al lector releer esto una y otra vez hasta que considere que realmente puede comprender los conceptos de puntero y dirección de memoria.

3.6.3 Argumentos por referencia

Para que una función pueda modificar el contenido de uno de sus argumentos tiene que recibir como parámetro su dirección de memoria.

Vamos a modificar la función `permutar` para que efectivamente pueda permutar los valores de los parámetros que recibe. Para que esto sea posible la función, en lugar de recibir dos parámetros de tipo `int` tendrá que recibir dos parámetros de tipo `int*`.

El prototipo será entonces:

```
void permutar(int*, int*);
```

y el código completo es el siguiente:

```
#include <stdio.h>

// prototipo de la funcion permutar
void permutar(int*,int*);

// programa principal
int main()
{
   int a=5,b=10;

   // muestro los valores de a y b
   printf("en main: a=%d, b=%d\n",a,b);

   // invoco a permutar y le paso las direcciones de memoria de a y b
   permutar(&a,&b);
```

```
   // muestro los valores de a y b
   printf("en main: a=%d, b=%d\n",a,b);

   return 0;
}

// funcion permutar, ahora recibe dos punteros a entero
void permutar(int* x,int* y)
{
   int aux;
   printf("...en permutarArgumentos x=%d, y=%d\n",*x,*y);

   // asigno el contenido de x a la variable aux
   aux = *x;

   // asigno el contenido de y al contenido de x
   *x = *y;

   // asigno el valor de aux al contenido de y
   *y = aux;

   printf("...en permutarArgumentos x=%d, y=%d\n",*x,*y);
}
```

Como vemos, ahora la función recibe dos parámetros de tipo `int*`. Las variables (o parámetros) `x` e `y` contienen direcciones de memoria. Dentro del cuerpo de la función, cuando hablamos de `*x` hacemos referencia al espacio de memoria direccionado por `x`.

Si `x` (de tipo `int*`) contiene la dirección de memoria en donde se aloja un valor de tipo entero entonces `*x` es el valor entero. Por este motivo, en los `printf` mostramos los valores de `*x` y `*y`.

Problema 3.3

Se dispone de un conjunto de valores que representan fechas expresadas como números enteros de 8 dígitos con formato *aaaammdd*.

Se pide informar:

1. ¿Cuántas fechas corresponden al mes de marzo?
2. ¿Cuántas fechas corresponden a años bisiestos?
3. Verificar que si en una fecha el día es 29 y el mes es 2 entonces que el año sea bisiesto. En caso contrario, mostrar un mensaje de error e informar, al final del proceso, la cantidad de veces que se registraron errores de este tipo.

Análisis

El problema en si mismo podría resolverse con tres contadores. Un contador para contar cuántas fechas corresponden a marzo, otro para contar cuántas fechas corresponden a años bisiestos y otro más para contar la cantidad de veces que se ingresan fechas con error.

Claro que para poder resolver el problema con tanta facilidad y naturalidad tendremos que abstraernos de los problemas puntuales que implican:

1. Dada una fecha, obtener su día, mes y año.
2. Dado un año, determinar si es bisiesto o no.

Para esto, diseñaremos (y luego desarrollaremos) las siguientes funciones:

`void dividirFecha(long, int*, int*, int*)`

Esta función recibe 4 parámetros. El primero es un entero de 8 dígitos que representa a una fecha con formato *aaaammdd*. El objetivo de la función es descomponer la fecha en día, mes y año y asignar cada uno de estos valores a los restantes parámetros.

`int esAnioBisiesto(int)`

Esta función recibe un entero que representa un año y retorna *true* o *false* (es decir, 1 o 0) según el valor del parámetro corresponda o no a un año bisiesto.
Contando con estas funciones, en la figura 3.5 podemos ver el algoritmo que resuelve el problema.

Con el problema resuelto, ahora tenemos que enfocarnos en los problemas particulares que implican: dividir la fecha en día, mes y año, y determinar si un año es bisiesto o no.
Estos dos problemas ya los analizamos en los capítulos anteriores así que aquí solo haremos un breve repaso.
Para obtener el año de una fecha representada en un número entero de 8 dígitos con formato *aaaammdd*, tenemos que realizar la división entera entre la fecha y 10000. Luego, el resto representa el mes y el día. Si utilizamos el valor residual para realizar una división entera por 100 obtendremos el mes y el resto de esta división corresponderá al día.
Respecto de determinar si un año es bisiesto o no, tenemos que verificar que sea múltiplo de 4 y que no lo sea de 100 o bien que sea múltiplo de 400.
El desarrollo de estas funciones lo podemos ver en la Fig. 3.6.

Ahora si, veamos el código fuente:

problema3.3.h

```
// prototipos
void dividirFecha(long, int*, int*, int*);
int esAnioBisiesto(int);
```

problema3.3f.c

```
void  dividirFecha(long f, int* d, int* m, int* a)
{
   int resto;

   *a=f/10000;
   resto=f%10000;
   *m=resto/100;
   *d=resto%100;
}

int esAnioBisiesto(int a)
{
   int multiploDe4,multiploDe100,multiploDe400;

   multiploDe4   = a%4==0;
   multiploDe100 = a%100==0;
   multiploDe400 = a%400==0;

   return (multiploDe4 && !multiploDe100) || multiploDe400;
}
```

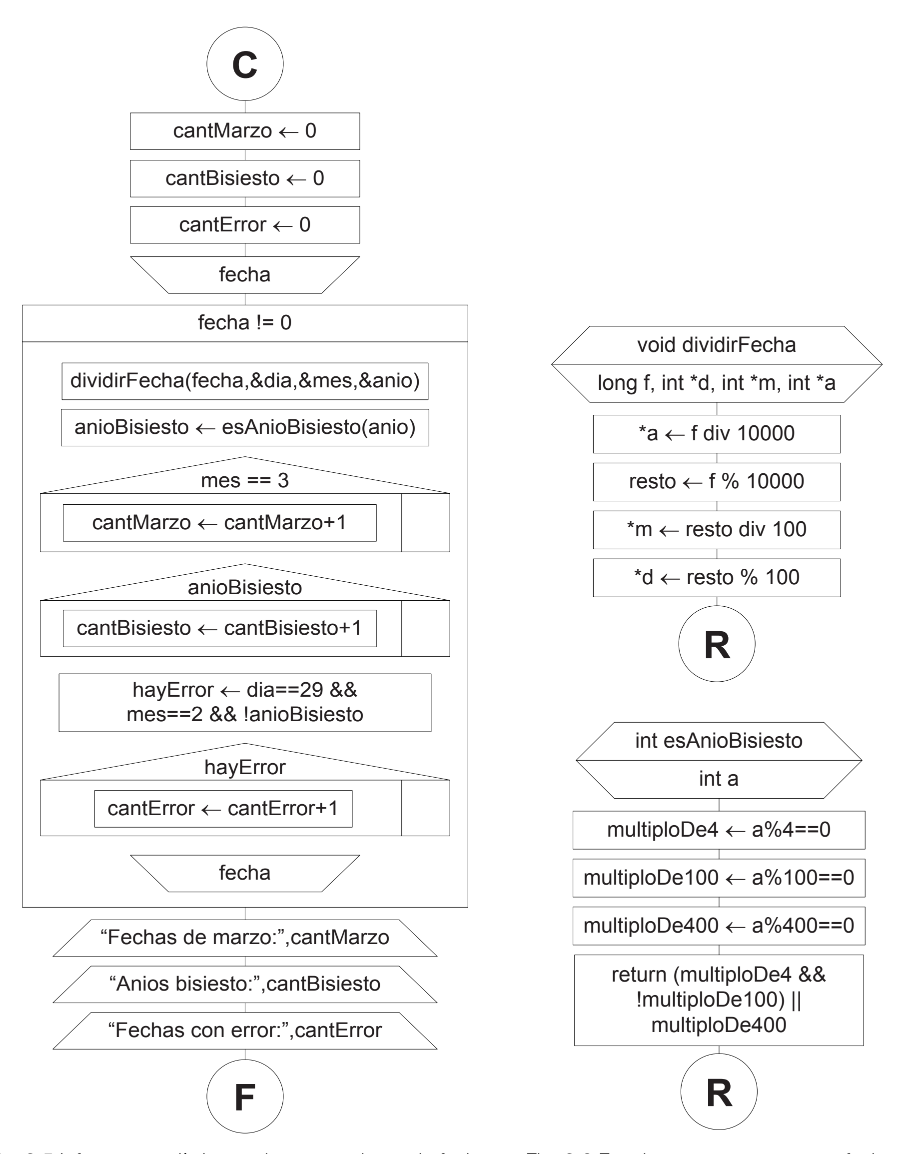


Fig. 3.5 Informa estadísticas sobre un conjunto de fechas. Fig. 3.6 Funciones para operar con fechas.

problema3.3p.c

```
#include <stdio.h>
#include "problema3.3.h"

// programa principal
int main()
{
   long fecha;
   int dia,mes,anio;
   int cantMarzo, cantBisiesto, cantError;
   int anioBisiesto, hayError;
```

```
   cantMarzo=0;
   cantBisiesto=0;
   cantError=0;

   printf("Fecha: ");
   scanf("%d",&fecha);

   while( fecha !=0 )
   {
      dividirFecha(fecha,&dia,&mes,&anio);
      anioBisiesto=esAnioBisiesto(anio);

      if( mes==3 )
      {
         cantMarzo=cantMarzo+1;
      }

      if( anioBisiesto )
      {
         cantBisiesto=cantBisiesto+1;
      }

      hayError = (dia==29) && (mes==2) && !anioBisiesto;

      if( hayError )
      {
         cantError=cantError+1;
      }

      printf("Fecha: ");
      scanf("%d",&fecha);
   }

   printf("Fechas de marzo: %d\n",cantMarzo);
   printf("Anios bisiesto: %d\n",cantBisiesto);
   printf("Fechas con error: %d\n",cantError);

   return 0;
}
```

Problema 3.3 bis

Mejorar el código desarrollado durante el problema 2.13 (página 67), invocando donde y cuando corresponda a la siguiente función:

```
void procesarBilletes(int* v, int denom)
{
   int cant = *v/denom;
   *v = *v%denom;
   printf("%d billetes de $%d\n",cant,denom);
}
```

Veamos el nuevo código, aplicando esta función:

```
int main()
{
   int v;

   // ingreso el sueldo a pagar
   printf("Ingrese el valor a pagar: $");
   scanf("%d",&v);

   procesarBilletes(&v,100);
   procesarBilletes(&v,50);
   procesarBilletes(&v,20);
   procesarBilletes(&v,10);
   procesarBilletes(&v,5);
   procesarBilletes(&v,2);
   procesarBilletes(&v,1);

   return 0;
}
```

El análisis de esta solución quedará a cargo del lector.

3.6.4 Funciones que mantienen su estado

En ocasiones necesitamos que una función pueda recordar el estado en el que quedó luego de haber sido invocada por última vez.

Por ejemplo, pensemos en desarrollar una función de modo que la primera vez que la invocamos retorne el primer número primo pero en la segunda invocación retornará el segundo y, genéricamente, en la *i-ésima* invocación retornará el *i-ésimo* número primo.

Evidentemente, esta función debe recordar cuántas veces fue invocada para poder determinar que número primo debe retornar o, pensándolo de otra forma, podría recordar cuál fue el último número primo que retornó para poder retornar el siguiente número primo en la próxima invocación.

Recordemos que no podemos admitir el uso de variables globales y, por otro lado, si intentamos mantener el estado en variables locales sabemos que estas se destruirán al finalizar la función y, en consecuencia, los valores que contienen se perderán.

Una solución válida podría ser que la función reciba un parámetro por referencia de forma tal que pueda almacenar en esta variable el valor del último número primo retornado.

Así, podemos desarrollar la función `siguienteNroPrimo` con el encabezado que se describe a continuación:

```
int siguienteNroPrimo(int* temp);
```

La función requiere que se le pase un parámetro por referencia donde depositará el último número primo que haya retornado. Luego utilizará este valor para, en la próxima invocación, poder retornar el primer número primo que sea mayor al valor de este parámetro.

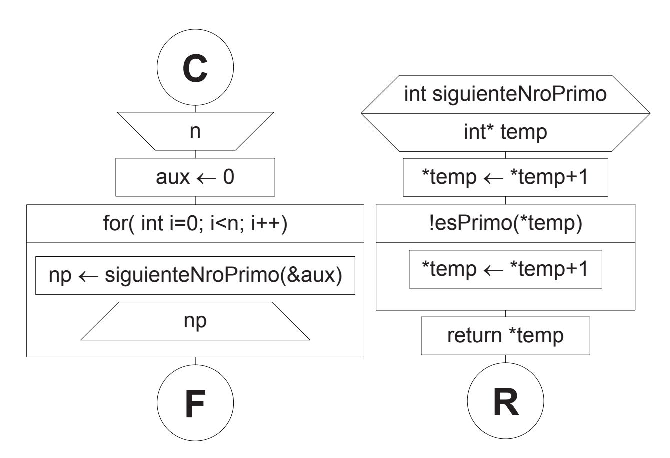


Fig. 3.7 Muestra los primeros *n* números primos.

En el gráfico vemos el desarrollo de la función `siguienteNroPrimo` y un programa que la utiliza para mostrar por consola los primeros `n` números primos.

En el programa simplemente invocamos `n` veces a la función y mostramos su valor de retorno (asignado previamente a la variable `np`).

Como la función requiere que le pasemos un parámetro por referencia, inicializamos en cero a la variable `aux` y le pasamos su dirección como argumento y, luego, "nos despreocupamos". La lógica para obtener el primer número primo, después el segundo, el tercero y así sucesivamente, es exclusiva responsabilidad de la función.

Analicemos ahora el algoritmo de la función donde, básicamente, la idea es la siguiente: obtener el primer número primo que sea mayor a `temp`. Así, si `temp` vale 0 incrementamos su valor mientras que este no sea primo. Luego, al encontrar el siguiente número primo, lo retornamos.

Notemos también que dentro de la función invocamos a la función `esPrimo` estudiada más arriba, en este mismo capítulo.

```
#include<stdio.h>

int siguienteNroPrimo(int* temp)
{
   *temp=*temp+1;

   while(!esPrimo(*temp))
   {
      *temp=*temp+1;
   }

   return *temp;
}

// programa principal
int main()
{
   int n;
   printf("Ingrese cuantos primos quiere ver: ");
   scanf("%d",&n);
```

```
   int aux=0;
   for(int i=0; i<n; i++)
   {
      printf("%d\n",siguienteNroPrimo(&aux));
   }

   return 0;
}
```

■

Problema 3.4

Desarrollar una función que reciba un valor numérico y que en cada invocación sucesiva retorne cada uno de sus factores. El valor de retorno de la función debe ser booleano para indicar si se pudo obtener un nuevo factor o no.

Análisis

Supongamos que a la función la llamamos `factorizar` y le pasamos como argumento el número 60. Entonces, según lo que pide el enunciado, deberíamos obtener los siguientes resultados:

`factorizar(n)`, con `n`=60

Invocación	**Factor**	**Valor de retorno**
1ra.	2	*true*
2da.	2	*true*
3ra.	3	*true*
4ta.	5	*true*
5ta		*false*

La función debe retornar dos valores: el factor de `n` que corresponda según la cantidad de veces que fue invocada y un valor booleano que indique si al llamar a la función se pudo obtener un nuevo factor o no.

Dado que las funciones solo pueden retornar un único valor tendremos que recurrir a un parámetro por referencia en el cual depositar el otro valor que queremos retornar.

Teniendo en cuenta esto, el encabezado de la función podría ser el siguiente:

```
int factorizar(int n, int* factor, int* temp);
```

La función retornará 1 o 0 (*true* o *false*) según se pueda o no seguir factorizando a `n`. Cada factor lo dejará en el parámetro `factor` y utilizará a `temp` como variable temporal para mantener el estado.

Con estas consideraciones, la estrategia para desarrollar el algoritmo de la función `factorizar` será la siguiente:

- Asignar `n` a `temp`.
- Incrementar `i` a partir de 2 y mientras que su valor no divida a `temp`.
- Cuando `i` divida a `temp` lo asignamos a `factor` y asignamos a `temp` el cociente `temp/i`.
- Cuando `temp` sea menor que 1 tendremos la certeza de que no se puede seguir factorizando a `n`.

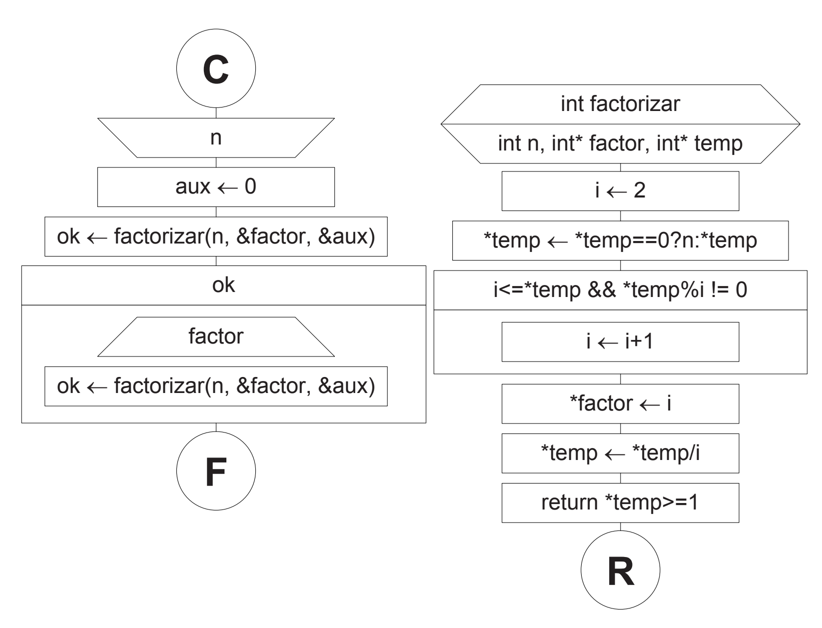


Fig. 3.8 Dado un valor numérico entero obtiene y muestra sus factores primos.

La estrategia elegida requiere que el valor inicial de `temp` sea igual a `n`. Para abstraer de este problema al programador que invoca a la función, lo primero que hacemos es asignar `n` a `temp` si el valor de `temp` es 0. Con esto, el parámetro auxiliar que se le debe pasar a `factorizar` puede venir inicializado en 0 o en `n`.

3.6.5 Variables estáticas (modificador `static`)

Como estudiamos más arriba, las variables locales de una función duran lo que dura su ejecución ya que cuando esta finaliza todas sus variables se destruyen.

El modificador `static` permite hacer que una variable local mantenga su valor entre las diferentes llamadas a la función.

Veamos un ejemplo: una función que utiliza una variable estática para retornar, en cada invocación, el siguiente número natural. Luego desarrollaremos un programa donde la invocaremos 10 veces para mostrar por consola los primeros 10 números naturales.

```
#include <stdio.h>

int siguienteNumero()
{
   // defino la variable estatica n, inicializada en 0
   static int n=0;


   // incremento el valor de n
   n=n+1;

   return n;
}
```

```
int main()
{
   // invoco 10 veces a la funcion
   for(int i=0; i<10; i++)
   {
      printf("%d\n", siguienteNumero() );
   }

   return 0;
}
```

La posibilidad de que una variable local mantenga su valor entre diferentes invocaciones a la función nos permite desarrollar versiones más simples y “amigables” de las funciones `siguienteNroPrimo` y `factorizar`.

Comencemos por analizar una nueva versión de la función `siguienteNroPrimo` donde utilizaremos una variable estática para recordar cuál fue el último número primo retornado por la función.

Con variable estática

```
int siguienteNroPrimo()
{
   static int temp=1;

   temp=temp+1;

   while( !esPrimo(temp) )
   {
      temp=temp+1;
   }

   return temp;
}
```

Sin variable estática (versión anterior)

```
int siguienteNroPrimo(int* temp)
{


   *temp=*temp+1;

   while( !esPrimo(*temp) )
   {
      *temp=*temp+1;
   }

   return *temp;
}
```

Como vemos, en la nueva versión, el valor del último número primo retornado se almacena en la variable estática `temp` y esta mantiene su valor entre las diferentes invocaciones a la función.

Ahora, podemos invocar a `siguienteNroPrimo` sin tener que pasarle ningún argumento auxiliar como vemos en el siguiente programa donde invocamos a la función para mostrar los primeros 10 números primos.

```
#include <stdio.h>

int main()
{
   for(int i=0; i<10; i++)
   {
      printf("%d\n", siguienteNroPrimo());
   }

   return 0;
}
```

Veamos ahora la nueva versión de la función `factorizar`:

```
int factorizar(int n, int* factor)
{
   static int temp=0;

   // si esta en cero le asigno n
   temp=temp==0?n:temp;

   int i=2;

   while( i<=temp && temp%i!=0 )
   {
      i=i+1;
   }

   *factor=i;
   temp=temp/i;

   return temp>=1;
}
```

3.7 Resumen

En este capítulo estudiamos, concretamente, como un problema extenso y complejo puede descomponerse en problemas más pequeños y simples. Este es el principio de la metodología *top-down* gracias al cual cualquier problema puede ser fácilmente resuelto independientemente de su nivel de complejidad.

En C los módulos se implementan como funciones. Aquí estudiamos los conceptos de argumentos y parámetros e invocación a funciones propias y de biblioteca.

También aprendimos que en C no existen los argumentos por referencia. Por este motivo, una función solo puede modificar los valores de sus parámetros si recibe sus direcciones de memoria en lugar de sus valores actuales.

Si bien el objetivo principal de este libro es el estudio de algoritmos y estructuras de datos, el hecho de haber optado por el lenguaje de programación C como lenguaje de implementación nos obliga a tratar algunos temas que son propios del lenguaje.

Quizás, uno de los puntos más crudos de este lenguaje de programación es la falta de un tipo de datos que nos permita manejar cadenas de caracteres. Ni en este capítulo ni tampoco en los anteriores este tema fue un impedimento para seguir avanzando, pero llegamos a un punto en el cual debemos analizar el tema con un mayor nivel de detalle.

En los siguientes capítulos, estudiaremos los tipos de datos alfanuméricos y la forma de implementar cadenas de caracteres en el lenguaje de programación C.

3.8 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

3.8.1 Mapa conceptual

3.8.2 Autoevaluaciones

3.8.3 Videotutorial

3.8.3.1 Mantener archivos de funciones separados del programa principal

3.8.4 Presentaciones*

4

Tipos de datos alfanuméricos

Contenido

Objetivos del capítulo

- Estudiar los tipos de datos alfanuméricos: carácter y cadena.
- Desarrollar funciones para manipular caracteres y cadenas de caracteres.
- Conocer las principales funciones que la biblioteca estándar de C provee para trabajar con cadenas de caracteres: `strcpy`, `strcmp`, `strcat`, `strlen`, `atoi`, `itoa`, `sscanf`, `sprintf`, entre otras.

Competencias específicas

- Conocer las características principales del lenguaje C.

4.1 Introducción

En el lenguaje de programación C, no existe un tipo de datos que permita contener cadenas de caracteres. Las cadenas se implementan sobre *arrays* de caracteres, tema que estudiaremos en detalle en el próximo capítulo.

Sin embargo, en lo que va del libro, intentamos abstraernos de este problema aceptando la idea de que un *array* de caracteres (o un `char[]`, léase char array) es un conjunto de variables de tipo `char`. Con esta premisa pudimos trabajar con cadenas aun sin haber estudiado *arrays*.

Hemos visto también que no podemos asignar directamente una cadena sobre un `char[]` sino que tenemos que hacerlo mediante una función que asigne uno a uno los caracteres de la cadena en las variables del conjunto.

En este capítulo, avanzaremos en el estudio de los datos alfanuméricos que incluyen el carácter y las cadenas de caracteres. Analizaremos la diferencia entre estos dos tipos y veremos las funciones de manejo de cadenas que se proveen en la biblioteca de C.

4.2 Carácter



ASCII *American Standard Code for Information Interchange* (Estándar para codificación de caracteres). Creado en 1963, de uso muy frecuente en el manejo de texto en computadoras y redes.

Llamamos carácter a todo símbolo tipográfico como ser una letra en mayúscula o en minúscula, un signo de puntuación, un dígito numérico, etc. Los caracteres están definidos en una tabla conocida como tabla ASCII en la que se le asigna a cada uno de ellos un valor o código numérico llamado código ASCII.

Según la tabla ASCII, el código numérico 65 corresponde al carácter ‘A’ y como la tabla respeta el orden alfabético resulta que el código 66 corresponde al carácter ‘B’, el código 67 representa al carácter ‘C’ y así sucesivamente. Análogamente, el código numérico 48 corresponde al carácter ‘0’, el código 49 representa al carácter ‘1’, etcétera.

Dado que los caracteres se representan a través de su código ASCII, se considera que son valores numéricos enteros y, a menudo, se los suele trabajar con el tipo de datos `int`. Sin embargo, existe un tipo de datos destinado específicamente para trabajar con caracteres: el tipo `char`.

4.2.1 El tipo de datos `char`

El tipo de datos `char` permite definir variables cuyo contenido será el código ASCII de un carácter. Como este código es un valor entero corto, el `char` es, en realidad, un tipo de datos numérico de 1 *byte* de longitud.

En C los caracteres se representan entre comillas simples. Por ejemplo, ‘A’, ‘B’, ‘5’, etc.

En el siguiente ejemplo, definimos una variable de tipo `char` y mostramos su valor expresado como carácter y luego como número.

```
#include <stdio.h>

int main()
{
   // asigno a c el valor 65 ('A' es sinonimo de 65)
   char c = 'A';
   printf("Como caracter: %c \n",c);
   printf("Valor numerico ASCII: %d\n",c);
```

```
    // asigno a i el valor 65
    int i = 'A';
    printf("Como caracter: %c \n",i);
    printf("Valor numerico ASCII: %d\n",i);

    return 0;
}
```

La salida de este programa será:

```
Como caracter: A
Valor numerico ASCII: 65
Como caracter: A
Valor numerico ASCII: 65
```

En el ejemplo vemos que podemos tratar de igual manera a las variables de tipo `int` y a las variables de tipo `char` porque, como se explicó más arriba, los caracteres son valores numéricos enteros.

4.2.2 Funciones para tratamiento de caracteres

Valiéndonos del hecho de que cada carácter es, en realidad, su código ASCII podemos definir una serie de funciones para operar con caracteres. Por ejemplo, una función que permita determinar si un carácter es un dígito numérico o una letra, una función para determinar si un carácter es una letra en mayúscula o en minúscula, etcétera.

4.2.2.1 Determinar si un carácter es un dígito numérico (función `esDigito`)

Definiremos la función `esDigito` que retornará *true* si el carácter que le pasamos como argumento es ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’ o ‘9’. En cualquier otro caso, retornará *false*.

Análisis

El algoritmo que resuelve el problema es trivial. La función simplemente debe retornar *true* o *false* según el carácter `c` que recibe como parámetro sea mayor o igual que ‘0’ y menor o igual que ‘9’.

```
int esDigito(char c)
{
    return c>='0' && c<='9';
}
```

4.2.2.2 Determinar si un carácter es una letra (función `esLetra`)

Esta función es análoga a la anterior, solo que retornará *true* o *false* dependiendo de que el argumento que se le pase sea o no una letra.

Análisis

La función debe determinar si el carácter `c` que recibe como parámetro corresponde a alguna de las letras del alfabeto. Esto es: retornará *true* si `c` está comprendido entre ‘A’ y ‘Z’ o entre ‘a’ y ‘z’. En cualquier otro caso, retornará *false*.

```
int esLetra(char c)
{
    return c>='A' && c<='Z' || c>='a' && c<='z';
}
```

4.2.2.3 Determinar si un carácter es una letra mayúscula o minúscula (funciones `esMayuscula` y `esMinuscula`)

Para que un carácter sea una letra mayúscula simplemente tiene que estar comprendido entre ‘A’ y ‘Z’. Análogamente, una minúscula estará entre ‘a’ y ‘z’.

```
int esMayuscula(char c)
{
   return c>='A' && c<='Z';
}

int esMinuscula(char c)
{
   return c>='a' && c<='z';
}
```

4.2.2.4 Convertir un carácter a minúscula (función `aMinuscula`)

Si el parámetro que recibe la función representa una letra mayúscula entonces retornará el mismo carácter, pero en minúscula. Si no simplemente retornará el mismo carácter que recibió. Veamos el código y luego lo analizaremos.

```
char aMinuscula(char c)
{
   return esMayuscula(c)?c-'A'+'a':c;
}
```

Análisis

Si `c` es mayúscula entonces la expresión `c-'A'` hace referencia a la posición que el carácter `c` ocupa dentro del alfabeto. Es decir, supongamos que `c` vale ‘B’ entonces su valor real es 66 ya que este es el valor de su código ASCII. La resta `c-'A'` en este caso será 66-65=1. Si a esto le sumamos ‘a’ cuyo código ASCII es 97 obtendremos 98 que corresponde al código ASCII del carácter ‘b’. Resumiendo, si `c` vale ‘B’ entonces:

c - ‘A’ + ‘a’ =
66 - 65 + 97 =
1 + 97 =
98 = ‘b’

4.2.2.5 Convertir un carácter a mayúscula (función `aMayuscula`)

Desarrollaremos ahora la función `aMayuscula` que es análoga a la función anterior. Es decir que si el carácter que recibe como parámetro representa a una letra en minúscula retornará el mismo carácter, pero en mayúscula.

Análisis

Si el carácter `c` que la función recibe como parámetro es una letra minúscula entonces con la resta `c-'a'` obtendremos la posición que la letra ocupa dentro del alfabeto y si luego le sumamos ‘A’ obtendremos la representación mayúscula del carácter que recibimos como parámetro.

```
char aMayuscula(char c)
{
   return esMinuscula(c)?c-'a'+'A':c;
}
```

Agrupemos los prototipos de todas estas funciones en el siguiente archivo:

`caracteres.h`

```
int esDigito(char);
int esLetra(char);
int esMayuscula(char);
int esMinuscula(char);
char aMinuscula(char);
char aMayuscula(char);
```

Ahora veamos el código de un programa que pone a prueba todas las funciones que desarrollamos:

```
#include <stdio.h>
#include "caracteres.h"

int main()
{
   char c='A';
   printf("esDigito(%c) = %d\n",c,esDigito(c));
   printf("esLetra(%c) = %d\n",c,esLetra(c));
   printf("esMayuscula(%c) = %d\n",c,esMayuscula(c));
   printf("esMinuscula(%c) = %d\n",c,esMinuscula(c));

   printf("aMinuscula(%c) = %c\n",c,aMinuscula(c));
   printf("aMayuscula(%c) = %c\n",c,aMayuscula(c));

   return 0;
}
```

4.3 Cadenas de caracteres

Llamamos “cadena de caracteres” o simplemente “cadena” a toda secuencia finita y ordenada de caracteres consecutivos. Por ejemplo, “Hola”, “Amor” y “Av. del Libertador 1234, piso 5” son cadenas de caracteres ya que cada una se compone de una secuencia finita y ordenada de caracteres consecutivos.

En general, se considera que las cadenas de caracteres constituyen un tipo de datos por lo que muchos lenguajes de programación proveen el tipo *string* (“cadena” en inglés) con el que se pueden definir variables de este tipo.

En el lenguaje de programación C, no existe el tipo de datos *string*. Las cadenas de caracteres se definen como `char[]` lo que hace que el tratamiento de cadenas sea un poco más complicado en C que en otros lenguajes como Pascal o Java.

Para ilustrar este problema vamos a comparar dos segmentos de código Java y Pascal.

Código Java

```
public class DemoJava
{
   public static void
   main(String args[])
   {
      // defino dos string
      String s,x;

      // le asigno una cadena
      s = "Hola ";

      // asigno a x la concatenacion
      x = s+"que tal?";

      // muestro el resultado
      System.out.println(x);
   }
}
```

Código Pascal

```
// defino dos strings
var s,x:string[20];
begin

   // le asigno una cadena
   s := 'Hola ';

   // asigno a x la concat...
   x := s+'que tal?';

   // muestro el resultado
   writeln(x);
end.
```

Como vemos, tanto en Java como en Pascal podemos definir una variable de tipo *string* y asignarle una cadena utilizando el operador de asignación. Luego podemos "sumarle" otra cadena obteniendo así la concatenación de ambas y asignar este resultado en otra variable, también de tipo *string*.

Como en C las cadenas se implementan sobre `char[]` las operaciones de asignación y concatenación (entre otras) tendremos que programarlas explícitamente para asignar uno a uno los caracteres de la cadena en las variables del *array*.

4.3.1 El carácter '\0' (barra cero)

Cuando definimos un `char[]` para contener una cadena, estamos reservando una cantidad fija y finita de variables de tipo `char` en las cuales vamos a asignar uno a uno sus caracteres. Esta cantidad de caracteres es fija y, en principio, no podrá ser modificada.

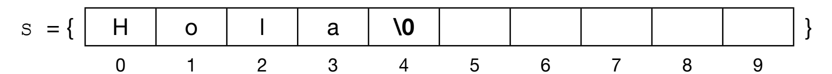
Como, generalmente, no podemos saber de antemano la cantidad exacta de caracteres que la cadena va a tener tendremos que acotarla. Por ejemplo, si necesitamos definir una variable para contener el nombre de una persona seguramente un `char[15]` será suficiente, pero si la variable fuera a contener el nombre y el apellido de la persona quizás necesitemos un `char[50]`. En cambio si el dato que vamos a almacenar en la variable será su dirección postal entonces necesitaremos contener más caracteres, tal vez un `char[150]`.

Lo anterior demuestra que la cantidad de caracteres de una cadena generalmente será menor que la dimensión o capacidad del `char[]` que la esté almacenando. Por lo tanto, será necesario delimitar el final de la cadena con un carácter especial: el '\0' (léase "barra cero").

Supongamos que definimos la variable `s` de la siguiente manera:

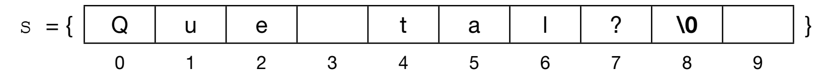
```
char s[10];
```

Entonces, si a `s` le asignamos la cadena "Hola" su representación interna será así:



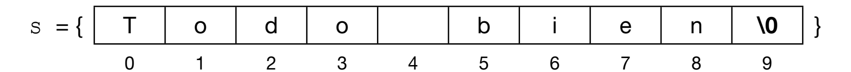
Aunque `s` puede contener 10 caracteres solo estamos utilizando cinco: cuatro para la cadena "Hola" más uno para el carácter '\0'. Vemos que la longitud de la cadena "Hola" es menor que la capacidad del *array* `s` que la contiene.

En cambio si a `s` le asignamos la cadena "Que tal?" su representación interna será la siguiente:



Aquí, de los 10 caracteres que podemos almacenar en `s` solo estamos usando 9: ocho para la cadena "Que tal?" más uno para el '\0'. La capacidad del *array* `s` sigue siendo mayor que la longitud de la cadena que contiene.

Ahora, si a `s` le asignamos la cadena "Todo bien" se verá así:



En este caso, estamos utilizando los 10 caracteres que la capacidad del *array* `s` permite almacenar: nueve para la cadena "Todo bien" más uno para el '\0'.

¿Qué sucederá si intentamos asignarle a `s` la cadena "Todo muy bien"?

La cadena `s` tiene capacidad para contener 10 caracteres. La responsabilidad de no exceder esta capacidad es del programador. C no impedirá que un programa intente acceder más allá de la capacidad asignada a un *array*.

Por lo tanto, si los *bytes* posteriores a la cadena están disponibles de casualidad, el programa funcionará bien. De lo contrario, se estará intentando acceder a bloques de memoria reservados por otros programas.

4.3.2 Longitud de una cadena

Llamamos "longitud de una cadena" a la cantidad de caracteres que la componen.

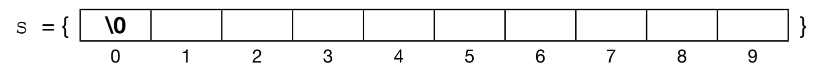
Así, la longitud de la cadena "Hola" es 4, la longitud de la cadena "Que tal?" es 8 y la longitud de la cadena "Todo bien" es 9.

Sin embargo, según lo que analizamos más arriba, para contener una cadena de longitud *n* será necesario utilizar un *array* con capacidad para almacenar al menos *n*+1 caracteres ya que tenemos que considerar el carácter '\0' para delimitar el final.

Es decir, una cadena de longitud *n* se almacena en *n*+1 caracteres y utiliza *n*+1 *bytes* de memoria.

4.3.2.1 La cadena vacía

Cuando una cadena tiene longitud igual a cero decimos que se trata de la cadena vacía y gráficamente podemos representarla de la siguiente manera:



Independientemente de la capacidad del *array* que estemos utilizando, la cadena vacía tiene longitud cero y su único carácter es el que delimita el final.

4.4 Tratamiento de cadenas de caracteres

Llamamos "tratamiento de cadenas de caracteres" a toda operación a través de la cual manipulamos los caracteres de una cadena. Por ejemplo: asignar una cadena a un `char[]`, concatenar dos o más cadenas, pasar cadenas a mayúsculas o a minúsculas, convertir cadenas a números, etcétera.

4.4.1 Inicialización de una cadena de caracteres

Diremos que una variable `s` definida como `char[]` es una cadena si y solo si contiene cero o más caracteres seguidos del carácter '\0'. De lo contrario, `s` solo será un *array* de caracteres, pero no podremos considerar que su contenido sea una cadena.

Cuando necesitamos trabajar con cadenas, definir variables de tipo `char[]` no es suficiente. Además, tenemos que asignarles valor inicial o "inicializarlas".

En el siguiente programa, veremos tres formas de inicializar cadenas de caracteres:

```
#include <stdio.h>

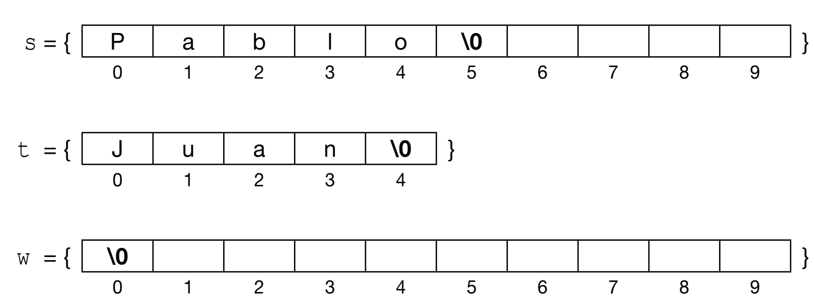
int main()
{
   char s[10] = "Pablo";
   char t[] = "Juan";
   char w[10] = { 0 };

   printf("s = [%s]\n",s);
   printf("t = [%s]\n",t);
   printf("w = [%s]\n",w);

   return 0;
}
```

En este código asignamos la cadena "Pablo" a la variable `s` cuya capacidad es de 10 caracteres. Luego definimos la variable `t`, pero no especificamos su capacidad por lo que el compilador dimensionará automáticamente al *array* con una capacidad suficiente como para contener los caracteres de la cadena "Juan" más 1 para el '\0'. Por último, definimos la variable `w` con capacidad para 10 caracteres y le asignamos `{ 0 }`. Esto la inicializará con la cadena vacía.

La representación gráfica de las cadenas `s`, `t` y `w` es la siguiente:



4.4.2 Funciones para el tratamiento de cadenas de caracteres

4.4.2.1 Asignar o copiar una cadena a un `char[]` (función `copiarCadena`)

El operador `=` (operador de asignación) únicamente puede utilizarse para inicializar cadenas en el momento de su definición. Si durante la ejecución de un programa necesitamos asignar una cadena a un `char[]` tendremos que hacerlo asignando uno a uno los caracteres de la cadena a los caracteres del *array*.

Definiremos entonces una función que reciba como parámetro dos `char[]` que llamaremos `t` y `s`. El objetivo será asignar uno a uno los caracteres de `s` en los caracteres de `t`. La capacidad de `t` debe ser mayor o igual a la longitud de `s` más 1.

Análisis

Como `s` contiene una cadena entonces su último carácter es '\0'. La estrategia para resolver la función será recorrer uno a uno los caracteres de `s` y, mientras no encontremos al carácter '\0', asignar el *i-ésimo* carácter de `s` al *i-ésimo* carácter de `t`.

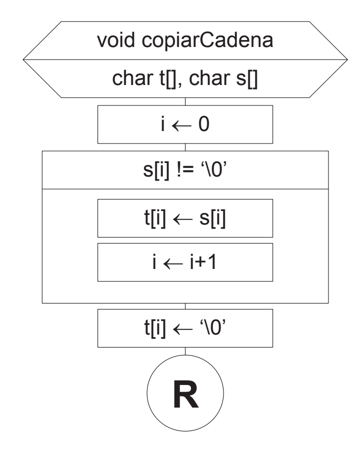


Fig. 4.1 Copia una cadena de caracteres.

Notemos que el `while` deja de iterar cuando `s[i]` vale '\0'. Por lo tanto, el carácter de fin de cadena tenemos que asignarlo manualmente una vez que hayamos salido del ciclo de repeticiones.

Veamos al código de la función y luego un programa principal que la utilice.

`cadenas.c`

```
void copiarCadena(char t[], char s[])
{
   int i=0;
   while( s[i]!='\0' )
   {
      t[i]=s[i];
      i=i+1;
   }

   t[i]='\0';
}
```

■

testCopiarCadena.c

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   char nom[15];

   // asigno la cadena "Pablo" a nom
   copiarCadena(nom,"Pablo");

   // muestro el contenido de nom
   printf("nom = %s\n",nom);

   return 0;
}
```

Recordemos que las cadenas literales, como en este caso la cadena “Pablo”, incluyen implícitamente el carácter ‘\0’ al final.

Notemos también que en el programa principal no fue necesario anteponer el operador de dirección `&` a `s` ya que, como estudiamos en los capítulos anteriores, un *array* es en sí mismo la dirección de memoria del primero de sus elementos. Por esto, cuando una función recibe un *array* como parámetro puede modificar su contenido.

4.4.2.2 Determinar la longitud de una cadena (función `longitud`)

Definiremos la función `longitud` que recibe un `char[]` y retorna la longitud de la cadena que contiene.

Análisis

Recordemos que las cadenas deben finalizar con el carácter ‘\0’. Por lo tanto, la estrategia para resolver esta función será recorrer la cadena y mientras no encontremos al carácter ‘\0’ incrementar un contador para contar cuantos caracteres contiene.

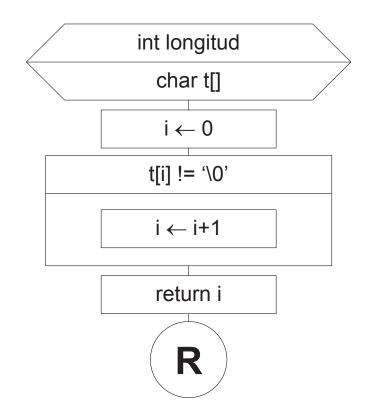


Fig. 4.2 Retorna la longitud de una cadena.

A continuación, veremos la codificación y luego un programa que la utiliza.

cadenas.c

```
// :
int longitud(char t[])
{
   int i=0;
   while( t[i] !='\0' )
   {
      i=i+1;
   }

   return i;
}
```

testLongitud.c

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   char nom[15];

   // asignamos una cadena a nom
   copiarCadena(nom,"Pablo");

   // mostramos el contenido de nom
   printf("nom = %s\n",nom);

   // mostramos la longitud de la cadena literal "Pablo"
   printf("longitud de %s = %d\n","Pablo",longitud("Pablo"));

   // mostramos la longitud de la cadena contenida en nom
   printf("longitud de %s = %d\n",nom,longitud(nom));

   return 0;
}
```

4.4.2.3 Determinar si una cadena es “vacía” (función `esVacia`)

Como comentamos más arriba, llamamos “cadena vacía” a la cadena de longitud cero que se origina cuando el primer carácter de un `char[]` es ‘\0’.

Definiremos entonces la función `esVacia` que recibe una cadena `s` y retorne *true* o *false* según `s` sea la cadena vacía o no.

cadenas.c

```
//:
int esVacia(char s[])
{
   return s[0]=='\0';
}
```

testVacia.c

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   char a[20] = { 0 };
   char b[20] = "Hola";
   char c[20] = "\0 que pasa ahora?";

   printf("a = [%s] es vacia? %d\n",a,esVacia(a));
   printf("b = [%s] es vacia? %d\n",b,esVacia(b));
   printf("c = [%s] es vacia? %d\n",c,esVacia(c));

   return 0;
}
```

En este ejemplo, las cadenas `a` y `c` son vacías. El lector no debe confundirse con el caso de la cadena `c`. Esta cadena tiene capacidad para 20 caracteres y fue inicializada con una serie de caracteres tal que el primero de ellos es '\0'. Este carácter delimita el final de la cadena, por lo tanto, los caracteres subsiguientes, si bien están contenidos en el *array* `c` no serán considerados como parte de la cadena.

4.4.2.4 Concatenar cadenas (función `concatenarCadena`)

Llamamos "concatenación" a la acción de agregar los caracteres de una cadena al final del último carácter de otra.

Definiremos la función `concatenarCadena` que recibe los parámetros `char t[]` y `char s[]` y agrega los caracteres de `s` al final del último carácter de `t`.

En este caso `t` debe ser una cadena con longitud mayor o igual a cero. Es decir que al menos debe contener el carácter '\0' y debe tener una capacidad suficiente que permita almacenar `longitud(t)+longitud(s)+1` caracteres.

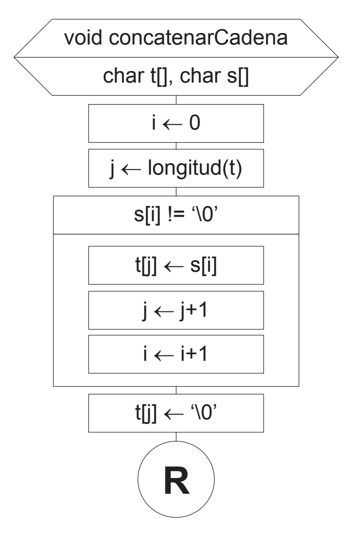


Fig. 4.3 Concatena dos cadenas.

Veamos la codificación:

cadenas.c

```
//:
void concatenarCadena(char t[], char s[])
{
   int i=0;
   int j=longitud(t);

   while( s[i]!='\0' )
   {
      t[j]=s[i];
      j=j+1;
      i=i+1;
   }

   t[j]='\0';
}
```

testConcatenarCadena.c

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   // defino e inicializo la variable x
   char x[20] = { 0 };

   // muestro que la longitud de x es cero
   printf("x=[%s], longitud(x)=%d\n",x,longitud(x));

   // concateno la cadena literal "Hola" a x
   concatenarCadena(x,"Hola");

   // muestro el contenido de x
   printf("x=[%s], longitud(x)=%d\n",x,longitud(x));

   // concateno la cadena literal " que tal?" a x
   concatenarCadena(x," que tal?");

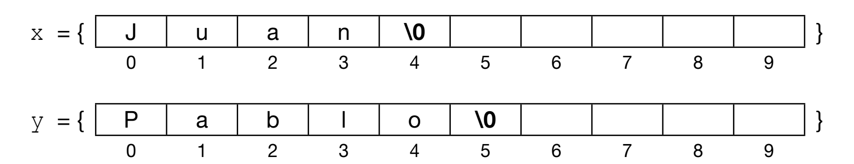
   // muestro el contenido de x
   printf("x=[%s], longitud(x)=%d\n",x,longitud(x));

   return 0;
}
```

4.4.2.5 Comparar cadenas (función `compararCadenas`)

Así como no podemos utilizar el operador de asignación `=` para asignar una cadena sobre un `char[]` tampoco podemos utilizar los operadores relacionales.

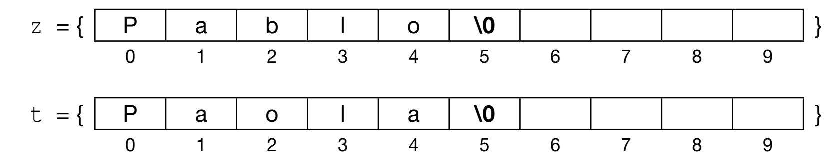
Para determinar si una cadena es alfabéticamente mayor, menor o igual que otra tenemos que comparar uno a uno sus caracteres. Analicemos los siguientes casos:



No hay dudas de que la cadena `y` es alfabéticamente mayor que la cadena `x` ya que "Pablo", al comenzar con 'P', resulta alfabéticamente mayor que "Juan" cuya inicial es 'J'. En este caso, analizar el primer carácter fue suficiente para determinar el orden de precedencia entre las cadenas `x` e `y`.

Sin embargo, ¿qué sucederá si ambas cadenas comienzan con el mismo carácter?

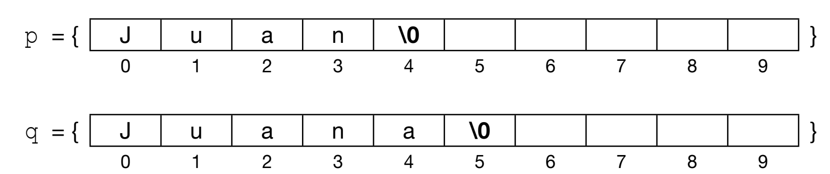
Veamos ahora las siguientes cadenas:



Aquí las cadenas `z` y `t` comienzan con la misma letra. Esto nos obliga a analizar el segundo carácter. Como este también es el mismo en ambas cadenas tendremos que analizar el tercero y, en este caso, podemos determinar que la cadena `t` es mayor que la cadena `z` porque el carácter 'o' es mayor que el carácter 'b'.

Es decir, sean las cadenas `a` y `b` diremos que `a` precede a `b` si `a[i]` es menor `b[i]` comenzando con `i=0` e incrementando su valor cada vez que `a[i]` sea igual a `b[i]`. Si para todo valor de `i` resulta que `a[i]` es igual a `b[i]` entonces diremos que ambas cadenas son iguales salvo que una de ellas tenga mayor longitud. En este caso, esta será la mayor.

Por ejemplo:



En este caso `q` es mayor que `p` porque si bien todos sus caracteres son idénticos su longitud es mayor.

Definiremos la función `compararCadenas` de la siguiente manera: recibe dos cadenas `a` y `b` y retorna un valor entero mayor, menor o igual a cero según `a` sea alfabéticamente mayor, menor o igual a `b`. Ver Fig. 4.4.

El algoritmo consiste en avanzar sobre los caracteres de ambas cadenas mientras sean iguales y, obviamente, mientras no llegue ningún '\0'. Cuando la condición del ciclo se deje de cumplir será porque el *i-ésimo* carácter de alguna de las dos cadenas es '\0' o es diferente del otro. Cualquiera sea el caso, la diferencia `a[i]-b[i]` será mayor, menor o igual a cero según el *i-ésimo* carácter de la cadena `a` sea mayor, menor o igual al *i-ésimo* carácter de la cadena `b`.

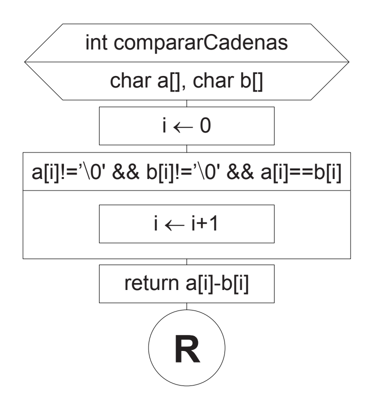


Fig. 4.4 Compara dos cadenas de caracteres.

Veamos la codificación:

`cadenas.c`

```
//:
int compararCadenas(char a[], char b[])
{
   int i=0;
   while( a[i]!='\0' && b[i]!='\0' && a[i]==b[i])
   {
      i=i+1;
   }

   return a[i]-b[i];
}
```

`testCompararCadenas.c`

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   char b[] = "Pablo";
   char a[] = "Juan";
   printf("%s vs. %s = %d\n",a,b,compararCadenas(a,b));

   return 0;
}
```

4.4.2.6 Convertir cadenas a números enteros (función `cadenaAEntero`)

En diversas ocasiones, tendremos cadenas cuyos caracteres solo serán dígitos numéricos y necesitaremos obtener un entero cuyo valor sea el número que está representado en dicha cadena. Por ejemplo, “123” es una cadena cuyos caracteres están representando al número 123.

Para analizar el algoritmo de una función, que nos permita obtener el número entero representado en una cadena de caracteres, estudiaremos la composición de los números enteros en el sistema decimal.

Por ejemplo, el número 859 indica una cantidad que se compone de 8 centenas, 5 decenas y 9 unidades. Por lo tanto, podemos descomponerlo y expresarlo como suma de productos de potencias de 10, la base de nuestro sistema numérico.

$$859 =$$
$$8x100 + 5x10 + 9 =$$
$$8x10^2 + 5x10^1 + 9x10^0$$

Podemos generalizar el algoritmo pensándolo como una suma de productos en la cual cada término se compone del producto entre el valor numérico del *i-ésimo* carácter de la cadena por $10^{n\text{-i-1}}$ donde *n* es la longitud de la cadena e *i* varia entre 0 y *n*.

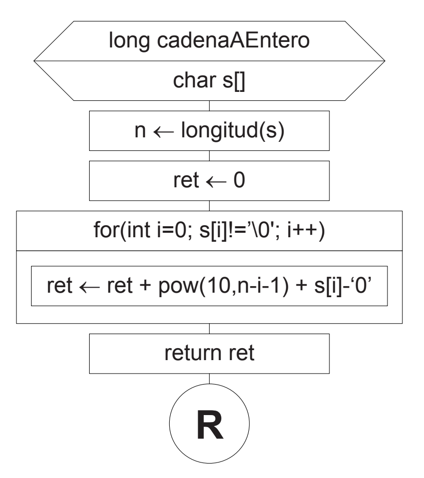


Fig. 4.5 Retorna el valor entero representado en una cadena.

Veamos la codificación de la función y un programa que la utiliza.

cadenas.c

```
#include <math.h>

// :
long cadenaAEntero(char s[])
{
   int n = longitud(s);
   double ret=0;

   for(int i=0; s[i]!='\0'; i++)
   {
      ret += pow(10,n-i-1) * (s[i]-'0');
   }

   return ret;
}
```

testCadenaAEntero.c

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   char s[]="12345";

   // asigno a n el valor numerico de s
   int n=cadenaAEntero(s);

   printf("Cadena = [%s]\n",s);
   printf("Numero = [%d]\n",n);

   return 0;
}
```

Si, en el algoritmo anterior, parametrizamos el valor de la base numérica podremos utilizarlo para obtener el valor decimal de cualquier cadena que represente un número expresado en base *n*, siendo *n* menor o igual que 10.

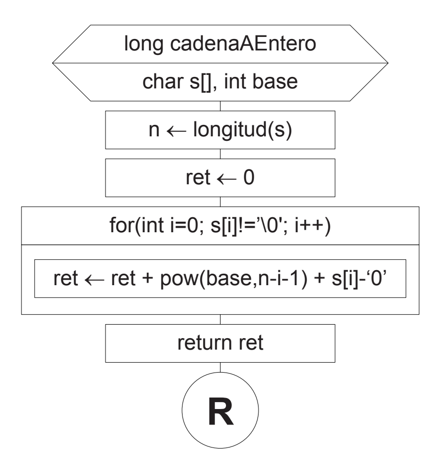


Fig. 4.6 Retorna el valor entero representado en una cadena, considerándolo en base *n*.

¿Qué modificaciones considera el lector que habría que realizarle a la función anterior para que permita, además, convertir cadenas que contengan números enteros expresados en base 16 o superiores? La codificación así como la modificación propuesta quedarán a cargo del lector.

4.5 Funciones de biblioteca para manejo de cadenas

La biblioteca de C incluye funciones que permiten copiar, concatenar, comparar y convertir cadenas. La mayoría de estas funciones están definidas en el archivo "string.h" y funcionan exactamente igual que las que hemos definido, analizado y programado en este capítulo.

Función de Biblioteca	Descripción	Análoga a
`strcpy`	Recibe dos `char[]` y asigna los caracteres que contiene el segundo al primero.	`copiarCadena`
`strcat`	Recibe dos cadenas y agrega los caracteres de la segunda al final de la primera.	`concatenarCadena`
`strlen`	Recibe una cadena y retorna su longitud.	`longitud`
`strcmp`	Recibe dos cadenas y retorna un valor mayor, menor o igual a cero según la primera sea mayor, menor o igual que la segunda.	`compararCadenas`
`atoi`	Recibe una cadena que contiene la representación de un valor numérico y un entero que indica una base y retorna un entero con el número representado en la cadena. Esta función está definida en el archivo "stdlib.h".	`cadenaAEntero`

4.5.1 Otras funciones de biblioteca

4.5.1.1 Dar formato a una cadena (función `sprintf`)

Esta función, definida en "stdio.h", funciona exactamente igual que `printf` solo que la salida la aplicará sobre un `char[]` que recibe como parámetro. Veamos un ejemplo:

```
#include <stdio.h>

int main()
{
   char nom[]="Pablo";
   int edad=39;
   double altura=1.70;

   char salida[50];

   sprintf(salida
          ,"Mi nombre es %s, tengo %d y mido %lf"
          ,nom
          ,edad
          ,altura);

   printf("%s\n",salida);

   return 0;
}
```

Como vemos, `sprintf` formatea la cadena y la asigna a `salida`. Luego, con `printf` mostramos su contenido por consola.

4.5.1.2 Interpretar (parsear) el formato de una cadena (función `sscanf`)

`sscanf` es la función inversa de `sprintf`. Recibe una cadena con un determinado formato y permite interpretarlo en función de los *placeholders* que especifiquemos en la máscara.

```c
#include <stdio.h>

int main()
{
   char cadena[]="Pablo 1.70 40";

   char nombre[10];
   float altura;
   int edad;

   sscanf(cadena,"%s %f %d",nombre,&altura,&edad);

   printf("Nombre: [%s]\n",nombre);
   printf("Altura: [%.2f]\n",altura);
   printf("Edad: [%d]\n",edad);

   return 0;
}
```

La salida de este programa es:

```
Nombre: [Pablo]
Altura: [1.70]
Edad: [40]
```

4.6 Resumen

En este capítulo, estudiamos la diferencia que existe entre los tipos de datos alfanuméricos: el carácter (`char`) y la cadena de caracteres (`char[]`).

Estudiamos y desarrollamos funciones utilitarias para, por ejemplo, determinar si un carácter es una letra o un dígito numérico, si está en mayúscula o en minúscula, etcétera. También desarrollamos funciones de tratamiento de cadenas de caracteres análogas a las que provee la biblioteca estándar de C.

Claro que, en todos los casos, en las funciones que desarrollamos trabajamos con cadenas cuya capacidad fue previamente definida en el programa invocador.

En el siguiente capítulo, veremos cómo se puede crear dinámicamente una cadena de caracteres, lo que nos permitirá analizar y desarrollar más funciones utilitarias.

4.7 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

4.7.1 Mapa conceptual

4.7.2 Autoevaluaciones

4.7.3 Presentaciones*

5

Punteros a carácter

Contenido

Objetivos del capítulo

- Comprender las diferencias y similitudes que existen entre *char[]* y *char**.
- Definir los conceptos de “cadena”, “subcadena”, “prefijo”, “sufijo”, “cadena vacía”, etcétera.
- Asignar memoria dinámicamente mediante la función *malloc*.
- Desarrollar funciones que retornen cadenas de caracteres (*char**).
- Conocer función *strtok* y utilizarla para tokenizar cadenas de caracteres.

Competencias específicas

- Conocer las características principales del lenguaje C.

5.1 Introducción

Como explicamos en varias oportunidades, el lenguaje de programación C no provee un tipo de datos que permita definir cadenas o *string*. Las cadenas se implementan sobre *arrays* de caracteres, pero, como *arrays* es un tema que analizaremos más adelante, intentamos restarle importancia haciendo hincapié en el hecho de que un *array* es un “conjunto” de variables del mismo tipo de datos por lo que un *array* de caracteres representa a un conjunto de variables de tipo `char`.

Hemos visto también que las funciones pueden modificar el contenido de las cadenas que reciben como parámetro y explicamos que esto es posible porque el *array* representa a la dirección de memoria del primero de sus elementos.

5.2 Conceptos iniciales

A la introducción anterior, le agregaremos el siguiente dato: las variables o elementos del *array* tienen direcciones de memoria consecutivas.

Según esto, un *array* representa la dirección de memoria del primer elemento de un conjunto de variables del mismo tipo cuyas direcciones son consecutivas y, en particular, un `char[]` representa la dirección del primer carácter de una serie de variables de tipo `char` con direcciones de memoria consecutivas. Es decir que un `char[]` encaja (*matchea*) perfectamente en un `char*` y lo probaremos a continuación.

En el siguiente código, programamos dos funciones: una recibe un `char[]` y la otra recibe un `char*`. Luego, en el programa principal, definimos e inicializamos una cadena y se la pasamos como argumento a las funciones. Veremos que, aunque una recibe un parámetro de tipo `char[]` y la otra recibe uno de tipo `char*`, ambas le dan exactamente el mismo tratamiento de cadena.

```
#include <stdio.h>

// prototipos de las funciones
void recibeArray(char[]);  // recibe un char[]
void recibePuntero(char*); // recibe un char*

int main()
{
   char s[] = "Esta es una cadena";
   recibeArray(s);   // le paso la cadena s a la funcion recibeArray
   recibePuntero(s); // le paso la cadena s a la funcion recibePuntero

   return 0;
}

void recibeArray(char x[])
{
   printf("x = %s\n",x);
   printf("x[3] = %c\n",x[3]);
}

void recibePuntero(char* x)
{
   printf("x = %s\n",x);
   printf("x[3] = %c\n",x[3]);
}
```

La salida de este programa será la siguiente:

```
x = Esta es una cadena
x[3] = a
x = Esta es una cadena
x[3] = a
```

En el código del programa anterior, verificamos que `char[]` *matchea* con `char*`. Tanto `recibeArray` como `recibePuntero` pueden recibir un `char[]`.

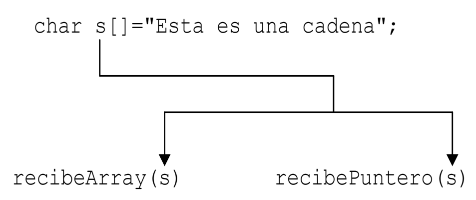


Fig. 5.1 Las funciones `recibeArray` y `recibePuntero` reciben el mismo argumento.

5.2.1 Aritmética de direcciones

Como el identificador de una cadena es equivalente a la dirección de memoria del primero de sus caracteres y las direcciones de los caracteres subsiguientes son consecutivas entonces la dirección el *i-ésimo* carácter se puede calcular sumando el valor entero *i* a dicho identificador.

Es decir, sea la variable `a` definida e inicializada de la siguiente manera:

```
char a[] = "Esto es una cadena";
```

`a` representa a la dirección de memoria del primer carácter de la cadena, `a+1` hace referencia a la dirección del segundo carácter y `a+i` hace referencia a la dirección de memoria del carácter ubicado en la posición *i*.

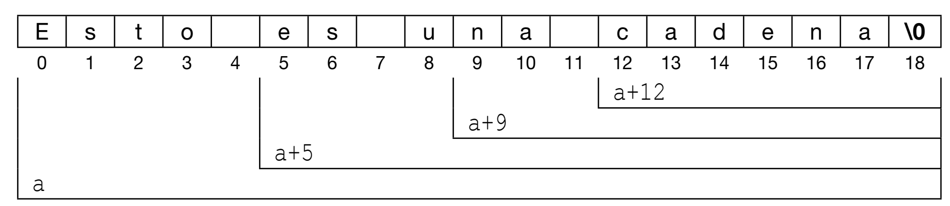


Fig. 5.2 Aritmética de direcciones.

En el gráfico vemos que al sumarle diferentes valores enteros al identificador de la cadena `a` obtendremos diferentes subcadenas comprendidas entre el *i-ésimo* carácter y el '\0', siendo *i* el valor entero que sumamos.

En el siguiente programa, definimos la cadena `a` e imprimimos algunas de las subcadenas que obtenemos luego de sumarle valores enteros a su identificador.

```
#include <stdio.h>

int main()
{
   char a[] = "Esto es una cadena";
```

```
    printf("[%s]\n",a);    // imprime: [Esto es una cadena]
    printf("[%s]\n",a+5);  // imprime: [es una cadena]
    printf("[%s]\n",a+9);  // imprime: [na cadena]
    printf("[%s]\n",a+12); // imprime: [cadena]

    return 0;
}
```

La salida será:

```
[Esto es una cadena]
[es una cadena]
[na cadena]
[cadena]
```

5.2.2 Prefijos y sufijos

Decimos que la cadena `p` es prefijo de la cadena `x` si los primeros caracteres de `x` coinciden con todos los caracteres de `p`. Análogamente, diremos que la cadena `s` es sufijo de la cadena `x` si los últimos caracteres de `x` coinciden con la totalidad de los caracteres de `s`.

Por ejemplo, sean las cadenas `x`, `p` y `s` definidas e inicializadas de la siguiente manera:

```
char x[] = "Esto es una cadena";
char p[] = "Es";
char s[] = "dena";
```

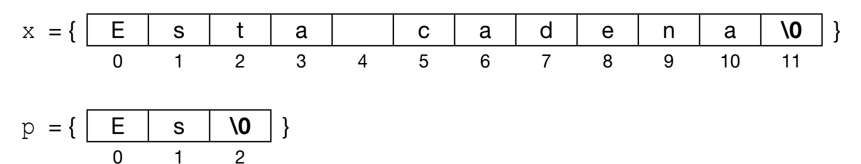
Según lo expuesto más arriba, `p` es un prefijo de `x` y `s` es un sufijo. Otros prefijos de `x` podrían ser: “Esto”, “E”, “Esto es una”. Otros sufijos de `x` serían: “cadena”, “a”, “na”, etcétera.

5.2.2.1 Determinar si una cadena es prefijo de otra (función `esPrefijo`)

Desarrollaremos una función para determinar si una cadena es prefijo de otra. La función `esPrefijo` recibirá dos cadenas `x` y `p`, y retornará *true* o *false* según `p` sea o no prefijo de `x`.

Análisis

Si la cadena `p` es prefijo de la cadena `x` entonces `x` debe comenzar con una secuencia de caracteres idéntica a `p` y su longitud debe ser mayor o igual que la longitud de `p`. Veamos:



Si llamamos `n` a la longitud de `p` entonces el algoritmo para resolver esta función consistirá en comparar los primeros `n` caracteres de ambas cadenas. Si son idénticos será porque `p` es prefijo de `x`.

Para programarla utilizaremos la función de biblioteca `strncmp` que es análoga a la función `strcmp`, pero recibe un parámetro extra que indica cuántos caracteres queremos comparar.

cadenas.c

```
#include <string.h>

// :
int esPrefijo(char* x,char* p)
{
   int n = strlen(p);
   return strncmp(s,p,n)==0;
}
```

Como `n` contiene la longitud de `p` la función `strncmp` comparará los primeros `n` caracteres de `x` con la totalidad de los caracteres de `p`. Entonces, si los primeros `n` caracteres de `x` y `p` son idénticos `strncmp` retornará cero, la expresión: `strncmp(x,p,n)==0` será verdadera y nuestra función retornará *true*.

testEsPrefijo.c

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   char a[] = "Esta cadena";
   char b[] = "Es";
   printf("[%s] es prefijo de [%s]? %d\n",b,a,esPrefijo(a,b));

   return 0;
}
```

La salida será:

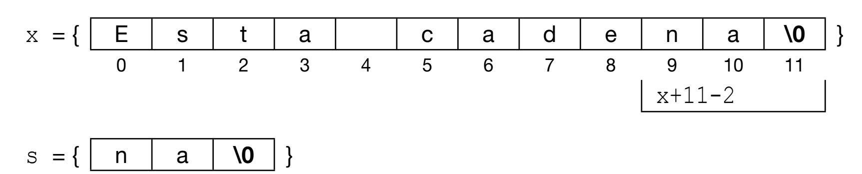
```
[Es] es prefijo de [Esta cadena]? 1
```

5.2.2.2 Determinar si una cadena es sufijo de otra (función `esSufijo`)

Desarrollaremos ahora una función que nos permitirá determinar si una cadena es sufijo de otra. La llamaremos `esSufijo`.

Esta función recibirá dos cadenas: `x` y `s` y retornará *true* o *false* según `s` sea o no sufijo de `x`. Para programarla, utilizaremos la función de biblioteca `strcmp` para comparar `s` con los últimos `n` caracteres de `x`, siendo `n` la longitud de `s`.

Supongamos que `x` es “Esta cadena” y `s` es “na” entonces, si llamamos `n` a la longitud de `s` podemos hacer referencia a los últimos `n` caracteres de `x` de la siguiente forma: `x+strlen(x)-n` que, según nuestro ejemplo, será: `x+11-2`.



Veamos el código de la función y luego un programa que la invoca.

cadenas.c

```
// :
int esSufijo(char* x,char* s)
{
   int desde=strlen(x)-strlen(s);
   return strcmp(s,x+desde)==0;
}
```

En este caso, invocamos a `strcmp` para comparar la cadena `s` con la subcadena de `x` comprendida entre el carácter `desde` y el '\0'.

testEsSufijo.c

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   char a[] = "Esta cadena";
   char b[] = "na";

   printf("[%s] es sufijo de [%s]? %d\n",b,a,esSufijo(a,b));

   return 0;
}
```

La salida será:

```
[na] es sufijo de [Esta cadena]? 1
```

5.3 Funciones que retornan cadenas

Hasta ahora hemos analizado funciones que manipulan los caracteres de las cadenas que reciben como parámetro, pero no analizamos ninguna función tal que su valor de retorno sea una cadena de caracteres.

En C las funciones no pueden retornar *arrays*, por lo tanto, sería incorrecto definir una función como la siguiente:

```
// ERROR, el valor de retorno de una funcion no puede ser un array
char[] obtenerSaludo();
```

Recordemos que una cadena es un `char[]` y su identificador representa la dirección del primero de una serie de caracteres con direcciones de memoria consecutivas. Así, la forma correcta de definir la función anterior es:

```
char* obtenerSaludo();
```

La función `obtenerSaludo` retorna un puntero al primer carácter de una serie de caracteres con direcciones de memoria consecutivas donde, además, el último será '\0'.

Sin embargo, todavía tenemos un problema y lo analizaremos con la siguiente implementación **errónea** de la función `obtenerSaludo`.

```
// implementacion erronea
char* obtenerSaludo()
{
   char a[] = "Hola, Mundo";
   return a;
}
```

El código anterior es semánticamente correcto ya que la cabecera de la función indica que se retornará un `char*` y, efectivamente, se retorna el *array* `a` que representa la dirección del primer carácter de la cadena “Hola, Mundo”.

Sin embargo, el *array* `a` es local a `obtenerSaludo` y, como sucede con cualquier variable local, será destruido al finalizar la ejecución de la última línea de esta función. Por lo tanto, el valor de retorno de `obtenerSaludo` será un puntero a un carácter que dejará de existir cuando el control regrese al programa o función que la invocó.

Si compilamos la función `obtenerSaludo` así como está, obtendremos el siguiente *warning* (advertencia) del compilador:

```
warning: function returns address of local variable
```

Notemos que este no es un error. Simplemente, es una advertencia porque, como explicamos más arriba, la codificación es semánticamente correcta.

Para que un `char[]` persista más allá de la función en donde fue definido tenemos que “crearlo” dinámicamente gestionando la memoria con la función de biblioteca `malloc` que analizaremos a continuación.

5.3.1 La función `malloc`

La función `malloc`, definida en el archivo “stdlib.h”, permite direccionar *n bytes* consecutivos de memoria siendo *n* un entero que le pasamos como argumento.

La memoria que gestionamos con `malloc` permanece asignada durante toda la ejecución del programa y trasciende a la función que la invocó.

Veamos la implementación correcta de la función `obtenerSaludo` analizada más arriba cuyo valor de retorno es una cadena de caracteres:

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

char* obtenerSaludo()
{
   // cadena local
   char a[]="Hola, Mundo";

   // longitud de la cadena que vamos a retornar
   int n = strlen(a);

   // array de n+1 caracteres generado dinamicamente
   char* r = (char*) malloc(n+1);

   // asigna la cadena al array gestionado dinamicamente
   strcpy(r,a);

   return r;
}

int main()
{
   // invoco a la funcion que retorna una cadena
   char* s = obtenerSaludo();

   // muestro la cadena
   printf("%s\n",s);

   return 0;
}
```

■

En esta implementación de `obtenerSaludo` invocamos a la función `malloc` para crear dinámicamente un `char[]` de `n+1` caracteres, donde `n` es la longitud de la cadena que queremos retornar más 1 para el carácter '\0'.

5.3.2 Subcadenas (función `substring`)

Dada la cadena `a` definida de la siguiente manera:

```
char a[] = "Esto es una cadena";
```

Llamamos "subcadena" a todo subconjunto de sus caracteres siempre y cuando estos sean consecutivos y mantengan el orden original.

Por ejemplo: "Esto es", "una cadena", "es", "a cad", "Est", "dena", "es una", "Esto es una cadena" y "" son algunas de las subcadenas que podemos obtener a partir de `a`. Notemos que todos los prefijos y sufijos de una cadena son también subcadenas de esta.

Desarrollaremos la función `substring` que recibirá una cadena y dos enteros indicando las posiciones `desde` y `hasta` (esta última será "no inclusive") y retornará la subcadena compuesta por los caracteres comprendidos entre dichas posiciones de la cadena original.

Por ejemplo:

- `substring("Esto es una cadena",0,4)` retorna "Esto".
- `substring("Esto es una cadena",8,11)` retorna "una".

La estrategia para resolver este algoritmo consistirá en calcular la longitud de la subcadena que tenemos que retornar, dimensionar dinámicamente un `char[]` de exactamente esa longitud (más 1 para incluir al carácter '\0') y luego asignar, uno a uno, los caracteres de la cadena original ubicados entre las posiciones `desde` y `hasta-1`.

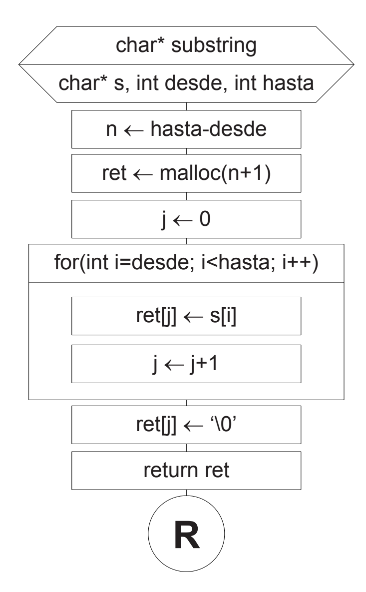


Fig. 5.3 Retorna una subcadena.

Veamos el código fuente:

cadenas.c

```
#include <stdlib.h>

//:
char* substring(char* s,int desde,int hasta)
{
   int n = hasta-desde;
   char* ret = (char*) malloc(n+1);

   int j=0;
   for(int i=desde; i<hasta; i++)
   {
      ret[j]=s[i];
      j=j+1;
   }

   ret[j]='\0';

   return ret;
}
```

testSubstring.c

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   char* x="Esto es una cadena";

   printf("%s\n",substring(x,0,4));  // imprime "esto"
   printf("%s\n",substring(x,8,11)); // imprime "una"

   return 0;
}
```

La salida del programa será:

```
esto
una
```

Problema 5.1

Analizando el código del siguiente programa. ¿Cuál es la diferencia que existe entre las cadenas `a` y `b`?

```
#include <stdio.h>
#include <string.h>
#include "cadenas.h"

int main()
{
   char s[50];
   char *a;
   char *b;
```

```
   strcpy(s,"Esto es una cadena");

   a = s+5;
   b = substring(s,5,strlen(s));

   printf("a = [%s]\n",a);
   printf("b = [%s]\n",b);

   return 0;
}
```

La salida del programa será la siguiente:

```
a = [es una cadena]
b = [es una cadena]
```

Análisis

Si bien ambas cadenas aparentan tener el mismo contenido la diferencia entre ellas es sustancial y para comprenderlo analizaremos el siguiente gráfico:

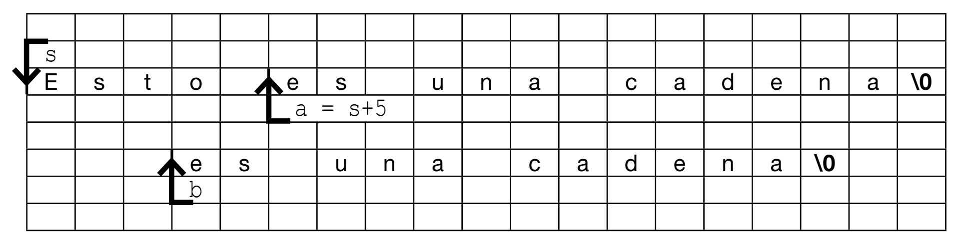


Fig. 5.4 Dos cadenas diferentes con la misma secuencia de caracteres.

Como vemos `s` y `a` están apuntando a diferentes caracteres de una misma cadena en cambio `b` apunta al primer carácter de una cadena diferente, independientemente de que el valor del *i-ésimo* carácter de `a` y `b` sea el mismo para todo *i*.

`a` es en realidad parte de la cadena `s` en cambio `b` es una cadena diferente y totalmente independiente. Cualquier modificación que apliquemos sobre `a` se verá reflejada en `s` y cualquier modificación que apliquemos sobre `s`, a partir de su carácter número 5, se verá reflejada en `a`. En cambio, las modificaciones que le hagamos a `b` serán independientes de `a` y de `s`.

Para probarlo modificaremos el programa anterior de la siguiente manera:

```
#include <stdio.h>
#include <string.h>
#include "cadenas.h"

int main()
{
   char s[50];
   char *a;
   char *b;

   strcpy(s,"Esto es una cadena");

   a = s+5;
   b = substring(s,5,strlen(s));
```

```
   printf("s = [%s]\n",s);
   printf("a = [%s]\n",a);
   printf("b = [%s]\n",b);

   a[3] = 'X';  // modifico un caracter de a
   b[3] = 'Y';  // modifico un caracter de b
   s[5] = 'Z';  // modifico un caracter de s

   printf("s = [%s]\n",s);
   printf("a = [%s]\n",a);
   printf("b = [%s]\n",b);

   return 0;
}
```

Ahora la salida será:

```
s = [Esto es una cadena]
a = [es una cadena]
b = [es una cadena]
s = [Esto Zs Xna cadena]
a = [Zs Xna cadena]
b = [es Yna cadena]
```

Vemos que la modificación que aplicamos sobre `a` impactó en `s` y la modificación que aplicamos sobre `s` también afectó a `a`. Sin embargo, la modificación que realizamos sobre `b` no tuvo impacto sobre ninguna de las otras cadenas.

5.3.2.1 Eliminar los espacios ubicados a la izquierda (función `ltrim`)

Definiremos la función `ltrim` cuyo objetivo será retornar una cadena idéntica a la que reciba como parámetro, pero sin espacios en blanco a la izquierda. Por ejemplo:

- `ltrim("   Hola")` retorna `"Hola"`
- `ltrim("Hola   ")` retorna `"Hola   "`.

El algoritmo consistirá en recorrer la cadena original mientras que el *i-ésimo* carácter sea `' '`. Cuando el ciclo de repeticiones finalice será porque `i` contiene la posición del primer carácter de la cadena que resulte ser distinto de `' '`. Entonces retornaremos la subcadena comprendida entre `i` y la longitud de la cadena `s` que recibimos como parámetro.

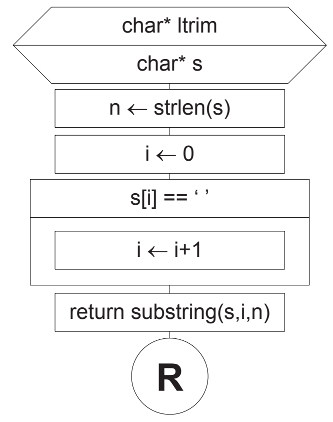


Fig. 5.5 Elimina los espacios ubicados a la izquierda de una cadena.

El código fuente es el siguiente:

cadenas.c

```
// :
char* ltrim(char* s)
{
   int n=strlen(s);
   int i=0;
   while(s[i]==' ')
   {
      i=i+1;
   }

   return substring(s,i,n);
}
```

testLTrim.c

```
#include <stdio.h>
#include "cadenas.h"

int main()
{
   char* x="      Esto es una cadena";
   printf("[%s]\n",ltrim(x));

   return 0;
}
```

La salida será:

```
[Esto es una cadena]
```

5.3.2.2 Eliminar los espacios ubicados a la derecha (función `rtrim`)

Esta función es análoga a la anterior solo que debe retornar una cadena sin espacios en blanco a la derecha.

La estrategia será recorrer la cadena comenzando desde atrás hacia adelante y mientras que encontremos espacios. Luego retornaremos la subcadena comprendida entre las posiciones 0 e *i*+1 de la cadena original.

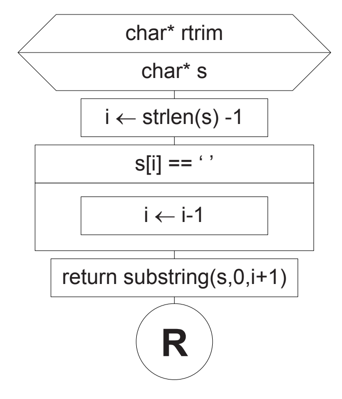


Fig. 5.6 Elimina los espacios ubicados a la derecha de una cadena.

El código es:

cadenas.c

```
//:
char* rtrim(char* s)
{
   int i=strlen(s)-1;
   while(s[i]==' ')
   {
      i=i-1;
   }

   return substring(s,0,i+1);
}
```

testRTrin.c

```
#include <stdlib.h>
#include "cadenas.h"

int main()
{
   char* x="Esto es una cadena     ";
   printf("[%s]\n",rtrim(x));

   return 0;
}
```

La salida será:

```
[Esto es una cadena]
```

5.3.2.3 Eliminar los espacios en ambos extremos de la cadena (función `trim`)

Como ya desarrollamos las funciones `ltrim` y `rtrim` para eliminar respectivamente los espacios ubicados a izquierda y a derecha de una cadena podemos pensar ahora en desarrollar la función `trim` para que retorne una cadena idéntica a la que reciba como parámetro, pero sin espacios en blanco en sus extremos. Esta función se programa fácilmente retornando el `ltrim(rtrim(s))` siendo `s` la cadena que recibe como parámetro.

cadenas.c

```
// :
char* trim(char* s)
{
   return ltrim(rtrim(s));
}
```

testRTrim.c

```
#include <stdio.h>
#include "cadenas.c"

int main()
{
   char* x="    Esto es una cadena     ";
   printf("[%s]\n",trim(x));

   return 0;
}
```

La salida será:

```
[Esto es una cadena]
```

5.3.3 Función de biblioteca `strtok`

Sea la cadena `s` definida en la siguiente línea de código:

```
char s[] = "Juan|Marcos|Carlos|Matias";
```

Si consideramos como separador al carácter '|' (léase "carácter paip") entonces llamaremos *token* a las subcadenas encerradas entre las ocurrencias de dicho carácter y a las subcadenas encerradas entre este y el inicio o el fin de la cadena.

Para hacerlo más simple, el conjunto de *tokens* que surgen de la cadena `s` considerando como separador al carácter '|' es el siguiente:

tokens = { "Juan", "Marcos", "Carlos", "Matias" }

Pero si en lugar de tomar como separador al carácter '|' consideramos como separador al carácter 'a' sobre la misma cadena `s` el conjunto de *tokens* será:

tokens = { "Ju", "n|M", "rcos|C", "rlos|M", "ti", "s" };

Utilizando la función `strtok` podemos separar una cadena en *tokens* delimitados por un separador. En el siguiente ejemplo, veremos cómo hacerlo:

```
#include <stdio.h>
#include <string.h>

int main()
{
   char s[]="Juan|Marcos|Carlos|Matias";
   char* tok;

   // primera llamada
   tok=strtok(s,"|");

   while( tok!=NULL )
   {
      printf("%s\n",tok);

      // llamadas subsiguientes
      tok=strtok(NULL,"|");
   }

   return 0;
}
```

La salida de este programa será:

```
Juan
Marcos
Carlos
Matias
```

Notemos `strtok` recibe como primer argumento la cadena que debe *tokenizar*. Luego, en cada invocación sucesiva, retorna el siguiente *token*. Evidentemente, `strtok` utiliza variables estáticas que le permiten recordar la cadena y cuál fue el *token* que retornó en la última invocación.

La constante `NULL` definida en `stdio.h` representa el valor de una dirección de memoria nula.

Aplicaremos las funciones de biblioteca `strtok` y `atoi` para desarrollar una nueva versión del problema del cambio, estudiado en los capítulos anteriores, cuyo enunciado recordaremos a continuación.

Problema 5.2

Dado un valor entero que representa un sueldo a pagar, y un conjunto de denominaciones, que representan los valores nominales de los billetes disponibles; informar qué cantidad de billetes de cada tipo se necesitará utilizar para abonar dicho sueldo. Se debe dar prioridad a los billetes de mayor denominación.

Análisis

En los capítulos anteriores, resolvimos el problema *hardcodeando* las denominaciones de los billetes. Incluso, en la versión mejorada donde invocamos varias veces a la función `procesarBilletes`, una vez por cada denominación.

Aprovecharemos la función `strtok` para *parsear* (interpretar) una cadena que contenga las denominaciones, separadas entre sí por un carácter separador. Veamos:

```
char denominaciones[] = "100|50|20|10|5|2|1";
```

Luego, *tokenizando* esta cadena podremos acceder a cada una de las denominaciones y aplicando `atoi` a cada *token* tendremos cada uno de los valores nominales listos para procesar.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void procesarBilletes(int*, int);

int main()
{
   // declaramos una cadena con las diferentes denominaciones
   char denominaciones[] = "100|50|20|10|5|2|1";
   int v;

   // ingreso el sueldo a pagar
   printf("Ingrese el valor a pagar: $");
   scanf("%d", &v);

   // tokenizamos la cadena y procesamos cada uno de sus tokens
   char* tok = strtok(denominaciones, "|");
   while( tok != NULL )
   {
      procesarBilletes(&v, atoi(tok));
      tok = strtok(NULL, "|");
   }

   return 0;
}
```

```
void procesarBilletes(int* v, int denom)
{
   int cant = *v / denom;
   *v = *v % denom;
   printf("%d billetes de $%d\n", cant, denom);
}
```

Si bien las denominaciones continúan *hardcodeadas* en nuestro programa, esta solución es mucho más flexible y extensible que las anteriores porque, si se llegase a agregar una nueva denominación o si dejase de existir cualquiera de las existentes bastará con aplicar estos cambios en la cadena `denominaciones`.

5.4 Resumen

En este capítulo utilizamos la función de biblioteca `malloc` para dimensionar dinámicamente cadenas (o *arrays*) de caracteres. Vimos también que, si bien el valor de retorno de una función no puede ser un *array*, bien puede ser la dirección de su primer elemento lo que, en definitiva, es la misma cosa.

Tuvimos un primer acercamiento a la aritmética de direcciones y estudiamos algunos conceptos de cadenas como ser: “prefijo”, “sufijo”, “subcadena” y *token*.

Todo esto, sumado a los conocimientos adquiridos en los anteriores capítulos, nos da las bases para poder estudiar con cierto nivel de detalle los conceptos de “dirección de memoria”, “puntero”, “*array*” y la relación que los une. Este será el tema del siguiente capítulo.

5.5 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

5.5.1 Mapa conceptual

5.5.2 Autoevaluaciones

5.5.3 Presentaciones*

6

Punteros, arrays y aritmética de direcciones

Contenido

Objetivos del capítulo

- Comprender los conceptos de "puntero" y "dirección de memoria".
- Analizar el funcionamiento de los operadores de dirección (&) e indirección (*).
- Aprender sobre el uso de *arrays* y analizar la relación que existe entre *arrays* y punteros.
- Aplicar aritmética de direcciones para desarrollar código fuente compacto y eficiente.
- Utilizar *arrays* de cadenas.
- Acceder a los argumentos pasados a través de la línea de comandos.

Competencias específicas

- Conocer las características principales del lenguaje C.
- Construir programas que utilicen arreglos unidimensionales y multidimensionales para solucionar problemas.

6.1 Introducción

En C los conceptos de “puntero” y *array* están totalmente ligados, razón por la cual forman parte de un mismo capítulo de estudio.

Si bien en los capítulos anteriores utilizamos *arrays* para implementar cadenas de caracteres, y utilizamos punteros para pasar argumentos por referencia a las funciones, aquí explicaremos en detalle cada uno de estos temas y la relación que los une.

En este capítulo, nos referiremos exclusivamente a cuestiones de implementación del lenguaje C, no explicaremos conceptos algorítmicos.

Debo prevenir al lector sobre la complejidad de los conceptos que vamos a tratar y recomendarle que, si la lectura se torna demasiado complicada, pase al capítulo siguiente y retome el presente cuando lo considere adecuado.

6.2 Punteros y direcciones de memoria

La memoria de la computadora se compone de un conjunto de *bytes* numerados consecutivamente. Luego, podemos almacenar datos en celdas de memoria formadas por grupos de 1 o más *bytes* según sea la longitud del dato que vayamos a almacenar.

En el Capítulo 1, estudiamos que en 1 *byte* podemos almacenar un carácter o un número entero comprendido entre 0 y 255 o entre -128 y 127, pero si necesitamos almacenar un número más grande necesitaremos utilizar celdas de 2 *bytes* o quizás más.

Supongamos que en un programa definimos las siguientes variables:

```
int a  = 10;  // 2 bytes
long b = 80;  // 4 bytes
char c = 'A'; // 1 byte
```

La declaración de estas variables hará que el programa, al comenzar a ejecutarse, reserve una cierta cantidad de *bytes* de memoria que dependerá de la longitud de sus tipos de datos. De esta forma, la distribución de la memoria de la computadora podría llegar a ser la siguiente:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	21	22	23	10		26	27	28	29	30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49	80				54	55	56	57	58	59
60	61	62	63		5	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	A	87	88	89	90	91	92	93	94	95	96	97	98	99

Fig. 6.1 Direccionamiento de la memoria.

El gráfico debe interpretarse de la siguiente manera: cada celda representa 1 *byte* de memoria cuya dirección está dada por un número secuencial.

Según el ejemplo, la dirección de la variable `a` es 24 y ocupa una celda compuesta por 2 *bytes*, la dirección de la variable `b` es 50 y utiliza 4 *bytes*. La variable `c` requiere un único *byte* y se ubica en la dirección de memoria número 86.

6.2.1 El operador de dirección `&`

Al anteponer el operador `&` (léase “ampersand”) al identificador de una variable obtenemos su dirección de memoria.

Siguiendo con el ejemplo anterior, el valor de la variable `a` es 10 y el valor de `&a` es su dirección de memoria que, en este caso, es 24. Veamos la siguiente tabla:

`a =`	10	`&a =`	24
`b =`	80	`&b =`	50
`c =`	‘A’	`&c =`	86

Fig. 6.2 Contenido de las variables vs. sus direcciones de memoria.

En la tabla, vemos el contraste entre el contenido de una variable y su dirección de memoria, la cual podemos obtener a través del operador `&`.

6.2.2 Los punteros

Llamamos “puntero” a una variable capaz de contener una dirección de memoria.

En las siguientes líneas de código, retomamos el ejemplo anterior y agregamos la variable `p` a la que le asignamos la dirección de memoria de la variable `a`. Luego diremos que “`p` es un puntero a `a`” o, simplemente, diremos que “`p` apunta a `a`”.

```
int a  = 10;  // 2 bytes
long b = 80;  // 4 bytes
char c = 'A'; // 1 byte

int* p = &a;
```

Como `a` es de tipo `int` entonces el tipo de datos de `p` es `int*` (léase “int asterisco” o “puntero a entero”). Los punteros o direcciones de memoria se representan en celdas de 4 *bytes*.

Ahora, considerando a la variable `p` la distribución de memoria podría ser la siguiente:

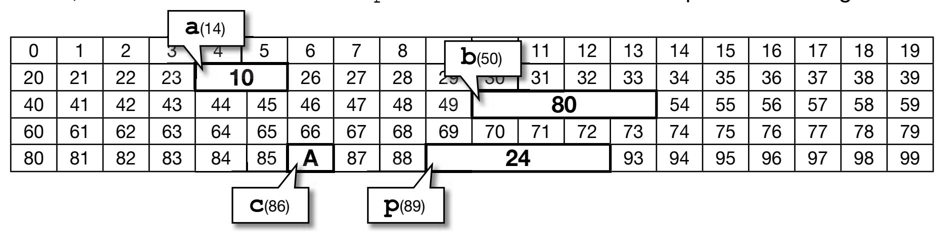


Fig. 6.3 Representación de una variable de tipo puntero.

Luego, si queremos acceder al espacio de memoria direccionado por `p` tendremos que hacerlo a través del operador de indirección `*` (léase “asterisco”) de la siguiente manera:

```
*p = 20;
```

En este caso, asignamos el valor 20 en el espacio de memoria direccionado por `p`. Claro que, como “`p` apunta a `a`”, indirectamente, la asignación la estaremos haciendo sobre esta variable.

6.2.3 El operador de indirección `*`

Como explicamos en el Capítulo 3, cuando hablamos de punteros el operador asterisco tiene doble utilidad:

1. Se usa para definir variables de tipo “puntero”.
2. Anteponiéndolo a un puntero podemos acceder al espacio de memoria que este direcciona.

6.2.4 Funciones que reciben punteros

También, en el Capítulo 3, explicamos que en C solo existen los parámetros por valor. De esta forma, las funciones trabajan con copias locales de los argumentos que le pasamos y cualquier modificación que realicen sobre estos tendrá alcance local.

Para que una función pueda modificar el valor de alguno de sus parámetros tiene que recibir su dirección de memoria.

Veamos nuevamente la función `permutar` que intercambia el valor de las variables que le pasamos como argumentos.

```
#include <stdio.h>

// prototipo
void permutar(int* ,int*);

int main()
{
   int a=5,b=10;
   permutar(&a,&b);
   printf("a=%d, b=%d\n",a,b);

   return 0;
}

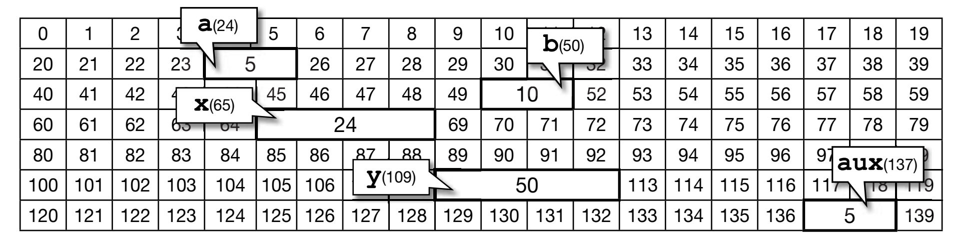
void permutar(int* x, int* y)
{
   int aux = *x;
   *x=*y;
   *y=aux;
}
```

En el programa principal, definimos las variables `int a=5,b=10` e invocamos a `permutar` pasándole sus direcciones de memoria: `&a` y `&b`. Dentro de la función, recibimos estas direcciones en los parámetros `int* x` e `int* y`.

Luego, en la primera línea de código de `permutar` definimos e inicializamos la variable `aux` con el valor direccionado por `x` haciendo:

`int aux=*x;`

En esta instancia el mapa de memoria podría ser el siguiente:



Luego, en el espacio de memoria direccionado por `x` asignamos el valor que está siendo direccionado por `y` haciendo: `*x=*y`. El mapa de memoria ahora será:

0	1			4	5	6	7	8	9	10		12	13	14	15	16	17	18	19
20	21			*10*		26	27	28	29	30			33	34	35	36	37	38	39
40	41	42			45	46	47	48	49	10		52	53	54	55	56	57	58	59
60	61	62		24				69	70	71	72	73	74	75	76	77	78	79	
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96			99
100	101	102	103	104	105	106			50				113	114	115	116			119
120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	5		139

Por último, en el espacio de memoria direccionado por `y` asignamos el valor de la variable `aux` con la siguiente línea de código: `*y=aux;`

0	1			4	5	6	7	8	9			12	13	14	15	16	17	18	19
20	21	22	23	10		26	27	28	29		31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49	5		52	53	54	55	56	57	58	59
60	61	62	63		24				69	70	71	72	73	74	75	76	77	78	79
80	81	82	83		85	86	87	88	89	90	91	92	93	94	95	96			99
100	101	102	103	104	105	106			50				113	114	115	116			119
120	121	122	123	124	125	126			129	130	131	132	133	134	135	136	5		139

a(24) b(50) x(65) y(109) aux(137)

Al terminar la función, sus variables locales `x`, `y` y `aux` se destruirán, pero los valores de las variables `a` y `b` quedarán modificados.

6.3 Arrays

Un *array* denota un conjunto de variables del mismo tipo cuyas direcciones de memoria son consecutivas.

```
int a[20];  // declara un array de 20 enteros
char c[10]; // declara un array de 10 caracteres
```

En estas líneas de código, el *array* `a` representa a un conjunto de 20 variables de tipo `int` y el *array* `c` representa a un conjunto de 10 variables de tipo `char`.

6.3.1 La capacidad del array

Llamamos “capacidad del *array*” a la cantidad de elementos o variables que lo componen.

En el ejemplo anterior, el *array* `a` tiene capacidad para almacenar 20 valores de tipo `int` mientras que la capacidad del *array* `c` permite almacenar 10 caracteres.

La capacidad de un *array* es estática, fija y se define en el momento de su declaración, no se puede modificar. Sin embargo, más adelante veremos que podemos crear un *array* dinámicamente invocando a la función de biblioteca `malloc`. En este caso, su capacidad podrá definirse durante la ejecución del programa.

6.3.2 Acceso a los elementos de un array

Una vez que el *array* ha sido definido (o declarado) disponemos de un conjunto de variables a las que podemos acceder a través del identificador del *array* más un subíndice (especificado entre corchetes) que indica la posición del elemento dentro del conjunto. La numeración siempre comienza desde cero.

Así, considerando los *arrays* `a` y `c` declarados más arriba, con las siguientes líneas de código asignamos valores en algunas de sus variables o elementos.

```
a[0] = 10; // asigno 10 en la primer posicion del array a
a[1] = 25; // asigno 25 en la segunda posicion de a

int x = 2;
a[x] = 30; // asigno 30 en la posicion x del array

c[0] = 'A'; // asigno 'A' en la primer posicion de c
c[1] = 66;  // asigno 66 (codigo ASCII de 'B') en c[1]
```

A continuación, analizaremos un programa muy simple en el que utilizamos un *array* para almacenar un conjunto de valores que ingresará el usuario por teclado.

El ejemplo es a título ilustrativo ya que, por el momento, no nos interesa profundizar en el estudio de algoritmos complejos. Como ya comentamos, el objetivo del capítulo es explicar los conceptos de punteros y *arrays* y exponer la relación que los une.

Problema 6.1

Desarrollar un programa que le permita al usuario ingresar un conjunto de 10 valores enteros. Luego se debe imprimir el conjunto que el usuario ingresó, primero en el orden original y luego, en orden inverso.

Por ejemplo, si el usuario ingresa: 12, 43, 5, 26, 7, 98, 1, 32, 18, 9 la salida del programa debe ser la siguiente:

```
Orden original: 12 43 5 26 7 98 1 32 18 9
Orden inverso: 9 18 32 1 98 7 26 5 43 12
```

Análisis

Para resolver este problema utilizaremos un `int[10]` (léase "array de 10 enteros") donde mantendremos los valores que ingresará el usuario. El primer valor lo almacenaremos en la primer posición del *array* (la posición número cero), el segundo valor lo almacenaremos en la segunda posición, y así sucesivamente.

Luego, para mostrar los valores ingresados debemos recorrer el *array* con un ciclo `for` cuya variable de control se incrementará entre 0 y 9 ya que, en cada una de estas posiciones se encuentran almacenados los elementos que el usuario ingresó.

Para mostrar los valores en orden inverso utilizaremos otro ciclo `for,` pero aquí la variable de control comenzará en 9 (último elemento del *array*) y se decrementará hasta llegar a 0 (primer elemento del *array*).

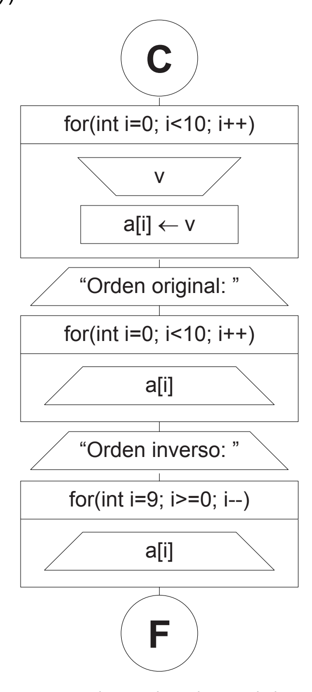


Fig. 6.4 Muestra un conjunto de valores al derecho y al revés.

La codificación es la siguiente:

```
#include <stdio.h>

int main()
{
   int v;
   int a[10];

   // leemos los datos que ingresa el usuario
   for(int i=0; i<10; i++)
   {
      printf("Ingrese un valor numerico (%d): ",i+1);
      scanf("%d",&v);

      a[i] = v;
   }

   // mostramos el conjunto en el orden original
   printf("Orden original: ");
   for(int i=0; i<10; i++)
   {
      printf("%d ",a[i]);
   }

   printf("\n");

   // mostramos el conjunto en el orden inverso
   printf("Orden inverso: ");
   for(int i=9; i>=0; i--)
   {
      printf("%d ",a[i]);
   }

   printf("\n");

   return 0;
}
```

6.3.3 Dimensionamiento e inicialización de arrays

Cuando declaramos un *array* estamos reservando celdas consecutivas de memoria, tantas como su capacidad, y el tamaño de cada una de estas celdas dependerá de su tipo de datos.

Es decir, si consideramos que para representar un `int` se necesita una celda de memoria de 2 *bytes* entonces un `int[10]` utilizará 10*2 = 20 *bytes*.

La declaración de un *array* solo implica reservar esa cantidad fija, finita y estática de celdas consecutivas de memoria. La responsabilidad de almacenar datos en cada una de estas celdas corre por cuenta del programador.

Sin embargo, existe la posibilidad de declarar un *array* especificando el conjunto de elementos que queremos que contenga. En este caso, su capacidad se dimensionará automáticamente en función de la cantidad de elementos del conjunto indicado.

Por ejemplo:

```
int a[] = {1, 2, 3, 4, 5};
char b[10] = {'\0'};
```

Aquí el *array* `a` tendrá capacidad para contener 5 enteros y cada uno de ellos será inicializado con el *i-ésimo* elemento del conjunto detallado entre llaves `{ }`.

En cambio, el *array* `b` tendrá la capacidad de contener 10 valores de tipo `char` de los cuales solo el primero se inicializará con el carácter '\0'.

Notemos que como el código ASCII del carácter '\0' es 0 entonces la declaración:

```
char b[10] = {'\0'};
```

es equivalente a:

```
char b[10] = {0};
```

6.3.4 Crear arrays dinámicamente (funciones `malloc` y `sizeof`)

Como estudiamos en el capítulo anterior, la función `malloc` permite asignar (o alocar) celdas contiguas de memoria y retorna un puntero con la dirección de la primera.

Es decir, sea la variable `sArr` de tipo `char*` y `n` de tipo `int` con algún valor entero positivo asignado, podemos crear un *array* de `n` caracteres de la siguiente manera:

```
sArr = (char*) malloc(n);
```

Luego, si hacemos `sArr[3]='A'` estaremos asignamos el carácter 'A' en la posición 3 del *array* `sArr`.

El tipo `char` siempre se representa en 1 *byte* de memoria. Sin embargo, la longitud de otros tipos de datos como `int` o `long` dependerá de la plataforma para la cual el programa fue compilado.

Para evitar problemas de portabilidad y asegurarnos que, independientemente de la plataforma con la que estemos trabajando, el *array* será correctamente dimensionado tenemos que utilizar la función `sizeof` como veremos a continuación.

Sean las variables `iArr` de tipo `int*` y `n` de tipo `int` con algún valor entero positivo, podemos crear un *array* de `n` enteros de la siguiente manera:

```
iArr = (int*) malloc(n*sizeof(int));
```

La función `sizeof` recibe un tipo de datos como parámetro y retorna la cantidad de *bytes* que ese tipo utiliza en la plataforma en la cual se está compilando el programa.

Volviendo al caso de los caracteres, `sizeof(char)` siempre retornará 1, por lo tanto, resulta indistinto invocar o no a `sizeof` al momento de asignar memoria para dimensionar un `char*`. Es decir que:

```
sArr = (char*) malloc(n*sizeof(char));
```

es equivalente a:

```
sArr = (char*) malloc(n);
```

6.3.5 Punteros genéricos `void*`

El tipo de datos `void` indica "nulidad". Tal es así que lo utilizamos para indicar que una función no retornará ningún valor.

En cambio, `void*` representa un puntero de cualquier tipo de datos. Es decir que los tipos `int*`, `char*`, `double*`, etc, son también (o encajan en) `void*`.

Sabiendo esto, podemos analizar el prototipo de la función `malloc` estudiada más arriba:

```
void* malloc(size_t size);
```

Como vemos, la función retorna un puntero cuyo tipo de datos es genérico, razón por la cual tenemos que convertirlo (*o castearlo*) al tipo de datos que esperamos recibir.

```
// convertimos a int* porque estamos asignando un array de tipo int
iArr = (int*) malloc(n*sizeof(int));

// convertimos a char*¨porque estamos asignando un array de tipo char
sArr = (char*) malloc(n*sizeof(char));
```

6.4 Relación entre arrays y punteros

Un *array* representa a un conjunto de celdas de memoria consecutivas y su identificador (la variable) es, en realidad, la dirección de memoria del primer elemento.

Es decir que, sea el *array* `a` definido como vemos a continuación:

```
int a[5] = {1, 2, 3};
```

podemos representarlo de la siguiente manera:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	2	a(68)	29	30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	1		2		3						78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99

En el gráfico vemos que el *array* se aloja a partir de la dirección de memoria 68, en 5 celdas consecutivas de 2 *bytes* cada una. La variable `a` es la dirección de memoria de la primera de estas celdas.

Resulta entonces que un `int[]` *matchea* (encaja) en un `int*` lo cual nos permite acceder al primer elemento del *array* a través de su dirección de memoria como vemos en las siguientes líneas de código:

```
int a[5];
int *p=a; // p apunta al primer elemento del array
*p = 10;  // equivale a hacer: a[0]=10
```

p(35)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	2	a(68)	29	30	31	32	33	34	68				39
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	1		2		3						78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99

6.4.1 Aritmética de direcciones

Sabemos que el identificador de un *array* es la dirección de memoria del primero de sus elementos. Entonces, ¿cuál será la dirección de memoria del segundo?

Siguiendo con el ejemplo del párrafo anterior, `p` es un puntero al primer elemento del *array* entonces la dirección de memoria del segundo elemento será `p+1` y, genéricamente, `p+`*i* será la dirección del elemento *i-ésimo*.

6.5 Código compacto y eficiente

La integración de punteros, *arrays* y aritmética de direcciones es uno de los puntos fuertes del lenguaje de programación C.

Cualquier operación que realicemos sobre *arrays* también podremos lograrla mediante la manipulación de punteros y para demostrarlo volveremos a analizar algunas de las funciones de tratamiento de cadenas de caracteres que estudiamos en los capítulos anteriores. Como las cadenas se implementan sobre `char[]` veremos que podemos recorrer y manipular sus caracteres accediéndolos a través de sus direcciones de memoria.

Además, veremos algunos operadores y posibilidades típicas del lenguaje C que, en combinación con la aritmética de direcciones, nos permitirán escribir código fuente compacto y altamente eficiente.

6.5.1 Operadores de incremento y decremento (operadores unarios)

El operador `++` suma 1 al valor de su operando. Análogamente, que el operador `--` le resta 1 a su valor.

En la siguiente tabla, comparamos el uso de estos operadores con la forma “tradicional” de incrementar y decrementar el valor de una variable.

Operadores `++` y `--`	**Incremento y decremento manual**
`int a=0;`	`int a=0;`
`a++`	`a = a+1;`
`// :`	`// :`
`a--;`	`a = a-1;`

Fig. 6.5 Uso de los operadores de incremento y decremento.

6.5.2 “Pre” y “post” incremento y decremento

Los operadores `++` y `--` pueden aplicase como prefijo o como sufijo de la variable que queremos incrementar o decrementar. En ambos casos, el operando resultará incrementado (o decrementado), pero la operación se realizará antes o después de evaluar su valor.

Veamos las siguientes situaciones:

```
int a=0;
a++; // ahora a vale 1
++a; // ahora a vale 2
printf("%d\n",a); // imprime 2
```

En casos como este, es lo mismo usar `a++` y `++a` para incrementar el valor de `a`. Sin embargo, C permite escribir sentencias compuestas por varias instrucciones y es en este tipo de situaciones donde tiene sentido hablar de “pre” o “post” incremento o decremento.

Veamos:

Postincremento	Preincremento

`int a=0;`
`printf("%d\n", a++);`
`printf("%d\n", a);`
 	
La salida será:
0
1
	`int a=0;`
	`printf("%d\n", ++a);`
	`printf("%d\n", a);`
	
	La salida será:
	1
	1

Fig. 6.6 Comparación entre “post” y “pre” incremento.

Como vemos, en el caso del postincremento `printf` utiliza el valor de `a` antes de que resulte incrementado por el operador `++`. Primero, utiliza su valor y luego lo incrementa.

En cambio, cuando hacemos `++a` (preincremento) el operador primero incrementa a la variable y luego `printf` utiliza su valor ya incrementado.

Veamos otro ejemplo:

Postincremento	Preincremento

`int a=0;`
 	
`if( a++ == 0 )`
`{`
`   printf("%d\n",a);`
`}`
 	
`printf("%d\n",a);`
 	
La salida será:
1
1
	`int a=0;`
	
	`if( ++a == 0 )`
	`{`
	`   printf("%d\n",a);`
	`}`
	
	`printf("%d\n",a);`
	
	La salida será:
	1

Fig. 6.7 Más comparaciones entre “post” y “pre” incremento.

En el primer caso, el `if` evaluará el valor de `a` sin incrementar, por lo tanto, la expresión lógica resultará verdadera. Luego, aún en la misma línea de código, el operador `++` incrementará a la variable por lo que, al ingresar al cuerpo del `if`, su valor será 1.

En cambio, en el segundo caso (preincremento) primero se incrementará el valor de `a` y luego el `if` evaluará su valor (ya incrementado). Esto hará que la expresión lógica resulte falsa. Por lo tanto, el programa no ingresará al cuerpo del `if`.

El operador `--` funciona de la misma manera que el operador `++`.

6.5.3 Operadores de asignación

Sea la variable `a` entera y con algún valor asignado, entonces `a++` incrementa su valor en 1 y `a+=n` lo incrementa en *n*.

Prácticamente, todos los operadores binarios (`+`, `-`, `*`, `/`, `%`, etc) tienen su correspondiente operador de asignación. Así, la expresión `a*=3` asigna a la variable `a` su valor actual multiplicado por 3 y la expresión `a%=2` asigna en esta variable el resto que se origina al dividir su valor por 2.

En la siguiente tabla, veremos algunos ejemplos:

Operadores de asignación	Asignación manual

`int a=0, n=6;`
`a+=5;`
`// :`
`a-=2;`
`// :`
`a+=n;`
	`int a=0, n=6;`
	`a = a+5;`
	`// :`
	`a = a-2;`
	`// :`
	`a = a+n;`

Fig. 6.8 Uso de los operadores de asignación.

6.5.4 Incremento de punteros

Cuando incrementamos el valor de un puntero hacemos que este apunte a la siguiente celda de memoria. Si sabemos que contamos con celdas de memoria consecutivas entonces esta posibilidad adquiere una gran importancia ya que nos permite desarrollar con punteros las mismas operaciones que realizamos sobre *arrays*.

6.5.4.1 Implementación compacta de la función `copiarCadena`

Para ilustrar esto estudiaremos el caso de la función `copiarCadena` que desarrollamos en el Capítulo 4. Recordemos su codificación:

```
void copiarCadena(char t[],char s[])
{
   int i=0;
   while( s[i]!='\0' )
   {
      t[i]=s[i];
      i++;
   }

   t[i]='\0';
}
```

Como ya sabemos, el identificador de un *array* equivale a la dirección de memoria del primero de sus elementos, por lo tanto, `char[]` *matchea* `char*`.

Por otro lado, como el código ASCII del carácter '\0' es 0 resulta que la expresión `s[i]!='\0'` es equivalente a `s[i]!=0` y, como en C los valores enteros tienen valor lógico y, en particular, el valor se considera falso entonces la expresión `s[i]!=0` equivale a decir: `s[i]`.

Apliquemos estas modificaciones a la implementación de la función:

```
// reemplazo los char[] por char*
void copiarCadena(char* t,char* s)
{
   int i=0;

   // mientras el i-esimo caracter sea verdadero (o mientras sea !='\0')
   while( s[i] )
   {
      t[i]=s[i];
      i++;
   }

   t[i]='\0';
}
```

Como `t` y `s` son punteros al primer carácter de sus respectivas cadenas entonces al incrementarlos pasarán a apuntar al carácter siguiente. Además `*t` es el carácter direccionado por `t`, es decir: `t[i]`. Análogamente `*s` es sinónimo de `s[i]`.
Veamos una nueva implementación de la función aplicando estos conceptos:

```
void copiarCadena(char* t,char* s)
{
   while( *s )
   {
      *t++=*s++;
   }

   *t='\0';
}
```

Como el `while` itera mientras que `s` no apunte al carácter '\0', cuando cuando llegue este carácter dejará de iterar y tendremos que asignar manualmente el carácter de fin de cadena en `*t`.
Sin embargo, podemos combinar el incremento de los punteros y la asignación del *i-ésimo* carácter con la condición del ciclo `while` y así lograremos copiar en `*t` también el carácter nulo.

```
void copiarCadena(char* t,char* s)
{
   while( *t++=*s++ );
}
```

Ahora comparemos la primera implementación de la función `copiarCadena` con esta última versión compacta:

copiarCadena (versión *array*)

```
void copiarCadena(char t[],char s[])
{
   int i=0;
   while( s[i]!='\0' )
   {
      t[i]=s[i];
      i++;
   }

   t[i]='\0';
}
```

copiarCadena (versión compacta)

```
void copiarCadena(char* t,char* s)
{
   while( *t++=*s++ );
}
```

6.5.4.2 Implementación compacta de la función `longitud`

longitud (versión *array*)

```
int longitud(char t[])
{
   int i=0;
   while( t[i] !='\0' )
   {
      i=i+1;
   }

   return i;
}
```

longitud (versión compacta)

```
int longitud(char* t)
{
   int i;
   for( i=0; *t++; i++ );
   return i;
}
```

6.6 Arrays de cadenas

Dado que una cadena es un `char*` entonces un *array* de cadenas debe implementarse como un *array* de punteros a carácter como se muestra en la siguiente línea de código:

```
char* arr[] = {"Hola", "Gracias", "Chau"};
```

Gráficamente, el *array* `arr` debería imaginarse así:

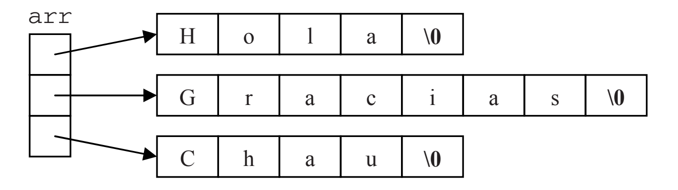


Fig. 6.9 *Array* de cadenas de caracteres.



The Beatles
Banda de rock inglesa de la década del sesenta, tal vez, la más popular de la historia. Extendieron su influencia a los movimientos sociales y culturales de su época.

En el siguiente programa, declaramos e inicializamos un *array* con los nombres de Los Beatles y luego mostramos su contenido.

```
#include <stdio.h>

int main()
{
   char* beatles[] = {"John", "Paul", "George", "Ringo"};

   for(int i=0; i<4; i++)
   {
      printf("%s\n",beatles[i]);
   }

   printf("All you need is love, love... love is all you need!\n");

   return 0;
}
```

Este programa es simple y no presenta ninguna complicación. Sin embargo, a continuación, veremos un programa que ilustra una situación típica en la cual el manejo de *arrays* de cadenas puede resultar confuso y engorroso.

Problema 6.2

Desarrollar un programa que le permita al usuario ingresar un conjunto de 5 cadenas de caracteres. Luego se debe imprimir el conjunto que el usuario ingresó, pero en orden inverso.

Análisis

Este problema es prácticamente igual al problema 6.1 que analizamos más arriba, solo que en lugar de trabajar con números enteros trabajaremos con cadenas.

La estrategia entonces será declarar un *array* que permita contener 5 cadenas y luego leer los datos que ingresará el usuario y asignarlos, uno a uno, en cada una de las posiciones del *array* para, luego, recorrerlo y mostrarlo.

La codificación que veremos a continuación es **incorrecta** e ilustra un error típico que suelen cometer los programadores que recién comienzan con el lenguaje C.

```
#include <stdio.h>

int main()
{
   char* cadenas[5];    // el array de cadenas
   char cadenaAux[10];  // una cadena auxiliar para leer

   for(int i=0; i<5; i++)
   {
      printf("Ingrese una cadena: ");
      scanf("%s",cadenaAux);

      cadenas[i] = cadenaAux; // ERROR
   }

   for(int i=4; i>=0; i--)
   {
      printf("%s\n",cadenas[i]);
   }

   return 0;
}
```

Supongamos que el usuario ingresa las siguientes cadenas en este orden: “Casa”, “Auto”, “Persona”, “Arbol” y “Sol”, entonces la salida del programa será:

```
Sol
Sol
Sol
Sol
Sol
```

Este resultado (salvo que estemos planificando unas vacaciones en la playa) no es el que esperábamos ver. El error se debe a que en cada iteración del `for` asignamos a `cadenas[i]` la dirección de `cadenaAux`:

```
cadenas[i] = cadenaAux; // ERROR
```

Cada vez que el usuario ingresa una cadena la almacenamos en `cadenaAux` sobrescribiendo su contenido anterior y luego hacemos que `cadenas[i]` apunte a `cadenaAux`. Así todas las posiciones de `cadenas` serán punteros a la misma y única `cadenaAux` y esta tendrá la última cadena de caracteres ingresada por el usuario que, según los datos del ejemplo, fue: “Sol”.

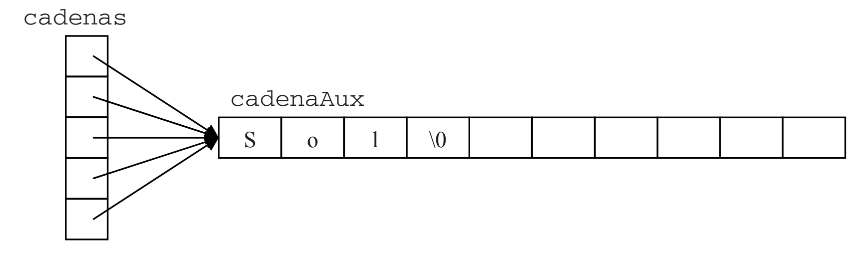


Fig. 6.10 *Array* de punteros a caracter, todos apuntando a una misma dirección.

Lo correcto será que cada posición (puntero) del *array* `cadenas` apunte a "su propia" cadena y que esta almacene el valor ingresado por el usuario.

Para esto, tendremos que, en cada iteración del `for`, crear una nueva cadena invocando a la función `malloc`, asignar su dirección a `cadenas[i]` y luego copiarle el contenido de `cadenaAux`.

```
// creo una nueva cadena y asigno su direccion a cadenas[i]
cadenas[i] = (char*) malloc(strlen(cadenaAux)+1);

// copio el contenido de cadenaAux a cadenas[i]
strcpy(cadenas[i],cadenaAux);
```

El resultado ahora será:

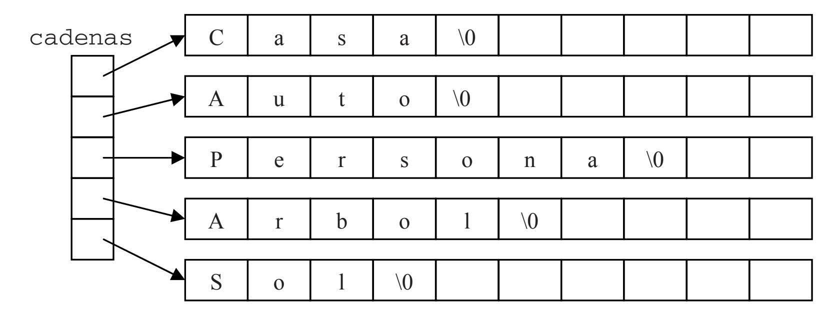


Fig. 6.11 *Array* de cadenas.

Veamos la codificación completa:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   char* cadenas[5];    // el array de cadenas
   char cadenaAux[10];  // una cadena auxiliar para leer

   for(int i=0; i<5; i++)
   {
      printf("Ingrese una cadena: ");
      scanf("%s",cadenaAux);

      // creo una nueva cadena y asigno su direccion a cadenas[i]
      cadenas[i] = (char*) malloc(strlen(cadenaAux)+1);

      // copio el contenido de cadenaAux a cadenas[i]
      strcpy(cadenas[i],cadenaAux);
   }

   for(int i=4; i>=0; i--)
   {
      printf("%s\n",cadenas[i]);
   }

   return 0;
}
```

Estas complicaciones son propias del lenguaje de programación C y no considero que sea necesario representarlas en un diagrama. Por lo tanto, en una representación gráfica podemos simplificar el problema y simplemente omitirlas como se muestra en los siguientes diagramas:

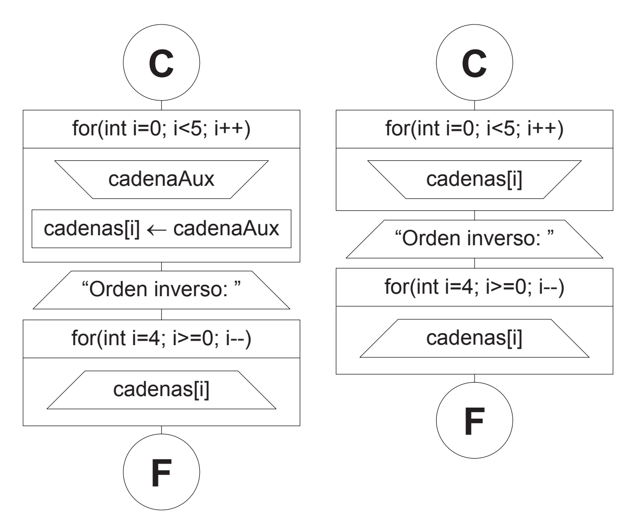


Fig. 6.12 Simplificación del manejo de cadenas, dos versiones.

6.6.1 Argumentos en línea de comandos (`int argc, char* argv[]`)

Podemos pasarle argumentos a los programas a través de la línea de comandos. Por ejemplo, cuando abrimos una consola de DOS y escribimos:

```
C:\>copy archivo1 archivo2
```

le estamos pasando al programa `copy` dos argumentos: las cadenas `archivo1` y `archivo2`.

Veamos otros ejemplos:

Programa	Argumentos
`format`	`c:`
`echo`	`"esto es una cadena"`
`tar`	`cvf pp.tar /home/pablosz`

En esta tabla vemos que al programa (o comando de DOS) `format` le pasamos como argumento la cadena `c:`. Al comando `echo` le pasamos la cadena `"esto es una cadena"` y al comando `tar` le pasamos 3 argumentos: `cvf`, `pp.tar` y `/home/pablosz`.

Desde el punto de vista del programa, los argumentos en línea de comandos siempre son cadenas de caracteres y se reciben en dos parámetros (opcionales) de la función `main`.

```
int main(int argc, char* argv[])
{
   //:
}
```



Pasar argumentos en línea de comandos con Eclipse



DOS (*Disk Operating System*, Sistema Operativo de Disco).
Fue el sistema operativo más popular para los procesadores Intel 8086 y 8088, de 16 bits.
La interfaz era una línea de comandos en modo texto o alfanumérico.

`argc` (*argument count*) es un entero que indica cuántos argumentos estamos recibiendo y `argv` (*argument vector*) es un *array* de cadenas que contiene `argc` elementos.

Debemos tener en cuenta que `argv[0]` siempre tiene el nombre del programa ejecutable por lo tanto, si a un programa le pasamos 3 argumentos en línea de comandos, el valor de `argc` será 4 y `argv[0]` será el nombre del programa ejecutable.

En el siguiente programa, mostramos por pantalla cada uno de los argumentos que recibimos a través de la línea de comandos.

muestraArgumentos.c

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   for(int i=0; i<argc; i++)
   {
      printf("%s\n",argv[i]);
   }

   return 0;
}
```

Luego, si lo invocamos de la siguiente manera:

```
C:\> muestraArgumentos hola que tal "todo bien?"
```

la salida será:

```
muestraArgumentos
hola
que
tal
todo bien?
```

Siguiendo con el ejemplo, el siguiente gráfico muestra que el *array* `argv` contiene 5 punteros a cadenas. La primera coincide con el nombre del programa, las restantes contienen los argumentos pasados en línea de comandos. En este caso, el valor de `agrc` será 5.

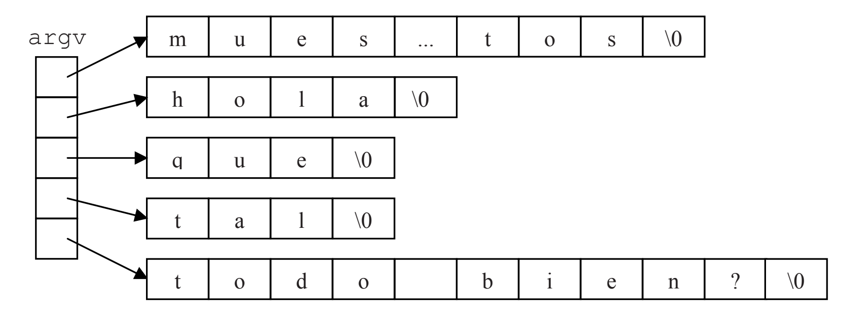


Fig. 6.13 Estructura del *array* `argv`.

6.7 Resumen

En este capítulo, estudiamos la relación que, en C, existe entre los punteros y los *arrays*. Un *array* representa un conjunto de celdas de 1 o más *bytes* con direcciones de memoria consecutivas, y su identificador es la dirección de memoria del primer elemento.

Partiendo de esta premisa vimos que, prácticamente, cualquier operación sobre *arrays* se puede resolver manipulando sus elementos a través de sus direcciones de memoria, lo que nos lleva a tener un código mucho más compacto y eficiente pero más difícil de leer.

Estudiamos también la estructura de un *array* de cadenas de caracteres y la forma en que un programa puede recibir parámetros a través de la línea de comandos.

En el siguiente capítulo, continuaremos estudiando *arrays*, pero desde el punto de vista algorítmico, analizando y desarrollando funciones utilitarias que nos permitirán agregar, eliminar y buscar elementos, ordenarlos, etcétera.

También estudiaremos el concepto de "tipo de dato estructurado": *arrays*, *arrays* cuyos elementos son *arrays* (matrices). Matrices cuyos elementos son *arrays* (cubos), estructuras, *arrays* de estructuras, etcétera.

6.8 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

6.8.1 Mapa conceptual

6.8.2 Autoevaluaciones

6.8.3 Videotutorial

6.8.3.1 Pasar argumentos en línea de comandos con Eclipse

6.8.4 Presentaciones*

7

Tipos de datos estructurados

Contenido

Objetivos del capítulo

- Analizar ejercicios que utilicen *arrays*.
- Comprender la diferencia entre acceso directo y acceso indirecto a los elementos de un *array*.
- Reforzar los conceptos de “capacitad” y “longitud” de un *array*.
- Desarrollar operaciones sobre *arrays*: buscar, agregar, insertar, eliminar, etcétera.
- Estudiar el algoritmo de ordenamiento por “burbujeo” (*bubble sort*) y el algoritmo de la búsqueda binaria o dicotómica (*binary search*).
- Trabajar con *arrays* multidimensionales: matrices y cubos.
- Crear nuevos tipos de datos con `typedef`.
- Entender la necesidad de usar registros o estructuras (`struct`).
- Lograr una primera aproximación al encapsulamiento a través de los TADs.

Competencias específicas

- Construir programas que utilicen arreglos unidimensionales y multidimensionales para solucionar problemas.
- Aplicar el método de búsqueda pertinente en la solución de un problema real.

7.1 Introducción

En los capítulos anteriores, trabajamos con tipos de datos simples y primitivos. Los tipos `int`, `long`, `short`, `char`, `void`, `float` y `double` son tipos primitivos ya que forman parte del lenguaje de programación, "vienen de fábrica". Además, son simples porque, por ejemplo, en una variable de tipo `int` podemos almacenar un único dato de ese tipo.

En cambio, los *arrays* representan conjuntos de variables de un mismo tipo de datos. Por lo tanto, decimos que un *array* es un dato de tipo estructurado.

En este capítulo, estudiaremos algoritmos que nos permitirán implementar operaciones sobre *arrays* a través de las cuales podremos manipular su contenido, y analizaremos tipos de datos estructurados ("registros" o "estructuras") definidos por el programador.

7.2 Acceso directo sobre arrays

Un *array* representa un conjunto de variables del mismo tipo, cada uno de ellos está identificado por el nombre del *array* (identificador) más un subíndice que indica la posición relativa de la variable dentro del conjunto.

Cuando trabajamos con conjuntos numéricos, acotados, consecutivos y "razonablemente" pequeños podemos utilizar a los elementos del conjunto como índice de acceso al *array* y así, establecer una relación unívoca entre estos elementos y las variables del *array*. En estos casos, hablamos de "acceso directo sobre el *array*".

Problema 7.1

Dado un conjunto de números enteros mayores o iguales a 0 y menores que 100 determinar e informar cuántas veces aparece cada uno. El conjunto finaliza con la llegada de un valor negativo.

Análisis

Antes de comenzar analizaremos un lote de datos tal como el que describe el enunciado.

```
A = {6, 2, 8, 1, 6, 1, 3, 2, 1, 2, 8, 3, 1}
```

Como vemos, todos los elementos del conjunto son valores numéricos mayores o iguales a cero y menores a 100 y algunos de ellos aparecen más de una vez. El 6 aparece 2 veces, el 2 aparece 3 veces, el 3 aparece 2 veces, el 1 aparece 4 veces y el 8 aparece 2.

El problema consiste en contar cuántas veces aparece cada número y para esto, necesitaremos mantener un contador por cada uno de los números que podría contener en conjunto `A`. Esto es: 100 contadores.

La única forma de trabajar con 100 contadores es implementarlos sobre un *array* de 100 elementos. Así, en la posición cero del *array* contaremos cuántas veces se repite el número cero. En la posición 1 del *array*, contaremos cuántas veces se repite el número 1 y, genéricamente, en la *i-ésima* posición del *array* contaremos cuántas veces se repite el número *i* siendo *i*>=0 y *i*<100.

Podemos ver que existe una relación unívoca entre cada elemento del conjunto `A` y el subíndice con el que identificamos a su contador: el contador asociado al número i está ubicado en la posición número *i* del *array*.

Veamos el algoritmo:

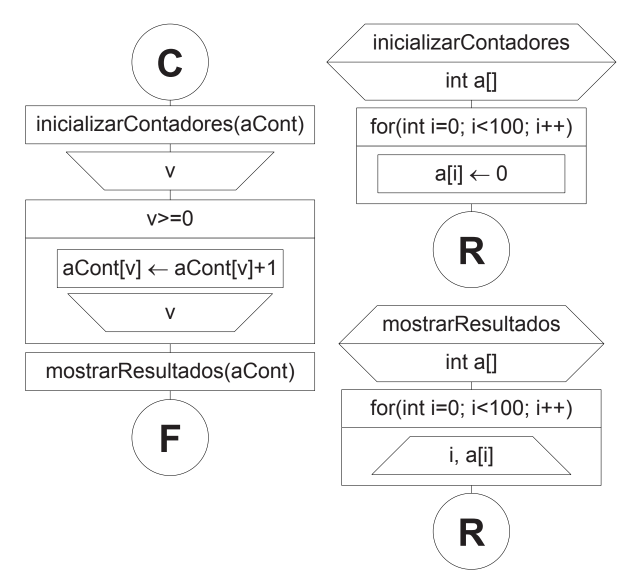


Fig. 7.1 Cuenta cuántas veces se repite cada número.

Como vemos, en el programa principal, hacemos un acceso directo a la posición `v` del *array* `aCont` ya que en esa posición se ubica el contador destinado a contar cuántas veces se repite el valor `v`.

Dentro de la función `mostrarResultados`, recorremos el *array* y mostramos su contenido. En esta función, `a[i]` es el contador que cuenta cuantas veces aparece el número `i` en el conjunto ingresado por el usuario. Por lo tanto, si consideramos que se ingresan los datos del conjunto `A` detallado más arriba, la salida del programa será la siguiente:

```
0    0
1    4
2    3
3    1
4    0
5    0
6    2
7    0
8    2
9    0
10   0
:    :
99   0
```

Otra posibilidad podría ser, mostrar solo aquellos valores que aparecen al menos una vez. En este caso, tendríamos que modificar la función `mostrarResultados` y antes de mostrar cada valor con su correspondiente contador preguntar si la cantidad de veces que apareció es mayor que cero.

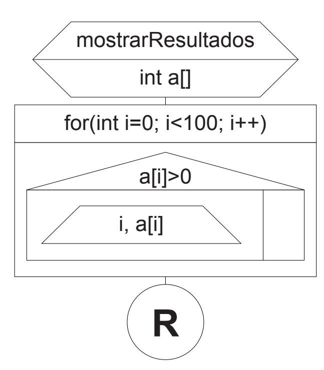


Fig. 7.2 Muestra solo aquellos valores que aparecen al menos una vez.

Con esta modificación e ingresando los datos del mismo conjunto `A`, la salida del programa será la siguiente:

```
1    4
2    3
3    1
6    2
8    2
```

Veamos la codificación:

problema7.1.h

```
void inicializarContadores(int a[])
void mostrarResultados(int a[])
```

problema7.1p.c

```
#include <stdio.h>

int main()
{
   int aCont[100];
   int v;

   // inicializo el array
   inicializarContadores(aCont);

   printf("Ingrese un valor: ");
   scanf("%d",&v);

   while(v>=0)
   {
      aCont[v]=aCont[v]+1;

      printf("Ingrese un valor: ");
      scanf("%d",&v);
   }

   // muestro los resultaods
   mostrarResultados(aCont);

   return 0;
}
```

problema7.1f.c

```
#include <stdio.h>

void inicializarContadores(int a[])
{
   for(int i=0; i<100; i++)
   {
      a[i]=0;
   }
}

void mostrarResultados(int a[])
{
   for(int i=0; i<100; i++)
   {
      if(a[i]>0)
      {
         printf("%d aparece %d veces\n",i,a[i]);
      }
   }
}
```

■

Problema 7.2

Se tiene una tabla que detalla la facturación emitida por un comercio durante el período de un mes, con el siguiente formato:

- `nroFactura` (número entero de 8 dígitos)
- `dia` (número entero entre 1 y 31)
- `importe` (número real)
- `codCliente` (alfanumérico, 5 caracteres)

Los datos no necesariamente están ordenados y puede haber más de una factura emitida en un mismo día. Finaliza cuando se ingresa un `nroFactura` igual a cero.

Se pide determinar e informar:

1. Total facturado por día, solo para aquellos días en los que hubo facturación.
2. Día de mayor facturación (será único) y monto total facturado ese día.

Análisis

Comenzaremos analizando un lote de datos que se ajuste al que describe el enunciado:

nroFactura	**dia**	**importe**	**codCliente**
10000345	3	150	AL321
10000360	1	300	ZN976
10000358	2	120	BQ322
10000321	1	50	ZN976
10000325	3	1000	XP967
10000343	4	250	BQ322
:	:	:	:

Según este lote de datos y de acuerdo con el enunciado del problema, vemos que los datos no respetan ningún orden. Podemos ver también que, por ejemplo, el día 3 se emitieron 2 facturas que suman un importe total de $1150 y, hasta el momento, este es el día de mayor facturación.

Para resolver este problema necesitaremos disponer de un acumulador por cada uno de los 31 días del mes y la única forma de lograrlo será utilizando un *array* con capacidad para 31 enteros. Lo llamaremos `acumDia`.

Nuevamente, existe una relación unívoca entre la posición del *array* y el valor acotado y consecutivo del conjunto de días en los que el comercio facturó. Es decir, los importes facturados el día 1 los acumularemos en el acumulador ubicado en la posición cero del *array*. Los importes facturados el día 2 los acumularemos en la posición 1 del *array* y, genéricamente, los importes facturados el día *i* los sumaremos en el acumulador ubicado en posición *i-1* del *array*. Recordemos que los *arrays* se numeran a partir de cero.

Según lo anterior, a medida que vayamos leyendo los datos de cada una de las filas de la tabla podremos acumular el importe de la factura en `acumDia[dia-1]` y así, al finalizar, tendremos en `acumDia[0]` el total facturado el día 1, en `acumDia[1]` el total facturado el día 2 y en `acumDia[i]` el total facturado el día `i+1` (para valores de `i` comprendidos entre 0 y 30).

Para determinar cuál fue el día de mayor facturación tendremos que esperar a tener la facturación total de cada día. Esto será una vez que hayamos leído y procesado todas las filas de la tabla. Entonces podremos recorrer el *array* y buscar en qué posición se encuentra el mayor valor. Si lo encontramos en la posición 5 será porque el día 6 fue el día que más se facturó, entonces en `acumDia[5]` tendremos el total facturado ese día.

Veamos el programa principal:

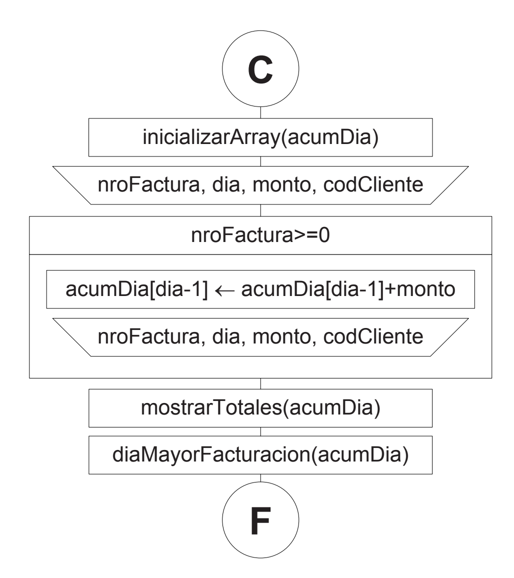


Fig. 7.3 Acumula el total facturado por día.

Dentro del ciclo de repeticiones, acumulamos el importe de la factura en `acumDia[dia-1]`. Esto hará que si `dia` es 1 entonces el importe se acumule en la primera posición del *array*. Y si `dia` es 2 entonces el importe se acumulará en la segunda posición.

Veamos ahora el desarrollo de las funciones `inicializarArray` en la que asignamos

0 a cada uno de los elementos del *array* y `mostrarTotales` en la que simplemente mostramos aquellos elementos del *array* cuyo valor es mayor que cero.

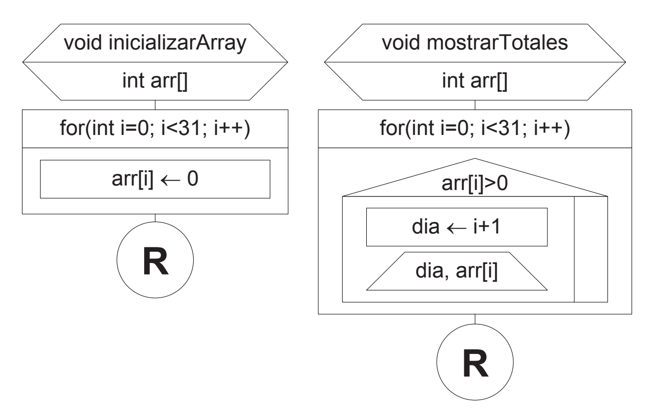


Fig. 7.4 Desarrollo de las funciones.

Por último, en la función `diaMayorFacturacion` recorremos el *array* para determinar en qué posición se encuentra el mayor valor y luego lo mostramos.

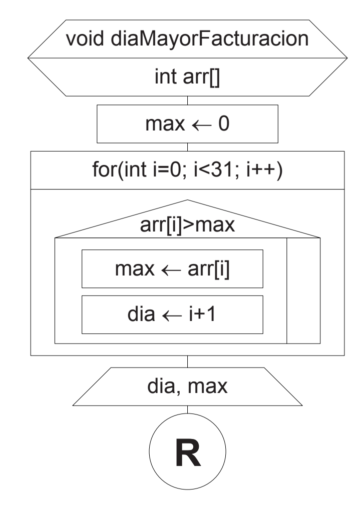


Fig. 7.5 Obtiene el mayor valor de los elementos de un *array*.

Veamos la codificación:

problema7.2.h

```
// prototipos de las funciones
void inicializarArray(double[]);
void mostrarTotales(double[]);
void diaMayorFacturacion(double[]);
```

■

problema7.2p.c

```
#include <stdio.h>

int main()
{
   // variables para lectura de datos
   long nroFactura;
   int dia;
   double importe;
   char codCliente[5];

   // array de acumuladores
   double acumDia[31];

   // inicializo el array
   inicializarArray(acumDia);

   // leo la primer fila de la tabla
   scanf("%ld %d %lf %s",&nroFactura,&dia,&importe,codCliente);

   while( nroFactura>0 )
   {
      // acumulo el importe facturado en el acumulador correspondiente
      acumDia[dia-1]=acumDia[dia-1]+importe;

      // leo la siguiente fila
      scanf("%ld %d %lf %s",&nroFactura,&dia,&importe,codCliente);
   }

   // muestro los totales facturados por cada dia
   mostrarTotales(acumDia);

   // determino y muestro el dia de mayor facturacion
   diaMayorFacturacion(acumDia);

   return 0;
}
```

problema7.2f.c

```
#include <stdio.h>

// inicializa el array asignando 0 a cada uno de sus elementos
void inicializarArray(double arr[])
{
   for(int i=0;i<31; i++)
   {
      arr[i]=0;
   }
}

// muestra los totales facturados por dia
void mostrarTotales(double arr[])
{
   int dia;

   printf("Totales facturados (dia, monto)\n");
```

```
   for(int i=0;i<31; i++)
   {
      if( arr[i]>0 )
      {
         dia=i+1;
         printf("%d, $%lf\n",dia,arr[i]);
      }
   }
}

// obtiene el dia de mayor facturacion y lo muestra
void diaMayorFacturacion(double arr[])
{
   double max=0;
   int dia;
   for(int i=0;i<31; i++)
   {
      if( arr[i]>max )
      {
         max=arr[i];
         dia=i+1;
      }
   }

   printf("Dia de mayor facturacion: %d, $%lf\n",dia,max);
}
```

■

7.3 Acceso indirecto sobre arrays

Hasta aquí analizamos casos en los que el mismo dato que estábamos procesando nos servía como índice para acceder al *array* pero, lamentablemente, esto no siempre será posible. Por ejemplo ¿qué sucedería si en el problema 7.1 los datos del conjunto no estuvieran acotados?

Reformulemos el problema para analizar esta nueva situación.

“Dado un conjunto de números enteros determinar e informar cuántas veces aparece cada uno. El conjunto finaliza con la llegada de un valor igual a cero y se garantiza que la cantidad de números diferentes que contiene el conjunto es de, a lo sumo, 100.

En este caso, sabemos que en el conjunto hay, a lo sumo, 100 números distintos, pero no sabemos cuáles son.

La lógica del algoritmo sigue siendo la misma: un contador para cada uno de los (a lo sumo) 100 diferentes valores del conjunto por lo que también utilizaremos un *array* de 100 posiciones para implementarlos, solo que no podremos establecer una relación directa entre cada una de las posiciones del *array* y cada uno de los valores del conjunto.

En este caso, tendremos que usar dos *arrays*, los llamaremos `aNum` y `aCont`. Cada vez que el usuario ingrese un valor `v` lo buscaremos en `aNum`. Si lo encontramos, supongamos, en la posición `i` entonces accederemos a `aCont[i]` para incrementar allí la cantidad de veces que `v` aparece en el conjunto. Si no lo encontramos será porque `v` es la primera vez que aparece así que lo agregaremos al final de `aNum` y en esa misma posición incrementaremos el valor de `aCont`.

Si bien la lógica del algoritmo sigue siendo la misma y su naturaleza es bastante simple, la implementación requiere accesos indirectos al *array* `aCont` lo que puede complicar las cosas.

Para solucionar situaciones como esta, analizaremos un conjunto de algoritmos básicos que nos permitirán fácilmente manipular los datos contenidos en *arrays*. Estos algoritmos agilizarán enormemente la resolución de problemas en los que se requiera el uso de *arrays* de acceso indirecto.

7.4 Operaciones sobre arrays

Las operaciones sobre *arrays* son algoritmos que nos permiten buscar, agregar, insertar y eliminar elementos. Analizaremos las siguientes operaciones:

- `agregar` - Permite agregar un elemento al final del *array*.
- `buscar` - Permite determinar si un *array* contiene un determinado elemento.
- `insertar` - Permite insertar un elemento en una determinada posición del *array*.
- `eliminar` - Permite eliminar el elemento ubicado en una posición determinada.
- `insertarEnOrden` - Inserta un elemento respetando un criterio de ordenamiento.
- `buscarEInsertarEnOrden` - Permite buscar un elemento y en caso de no encontrarlo entonces insertarlo respetando un criterio de ordenamiento.

Para simplificar y facilitar la lectura y la comprensión de los algoritmos trabajaremos con *arrays* de enteros (`int[]`), pero luego extenderemos estos conceptos para poder trabajar con *arrays* que contengan datos de cualquier otro tipo.

7.4.1 Capacidad vs. longitud de un array

Tal como lo definimos cuando estudiamos cadenas, haremos una diferenciación entre los conceptos de “capacidad” y “longitud” de un *array*.

Sea el *array* `a` de tipo `int[50]` entonces:

- La capacidad de `a` es: 50.
- La longitud de `a` en principio será 0 y aumentará en la medida en que vayamos agregando elementos al *array*. A la longitud del *array* la llamaremos `len` (abreviatura de *length*, “longitud” en inglés).

La longitud indica cuántos elementos están siendo contenidos por lo que si agregamos elementos al *array* debemos incrementar su longitud. En cambio, la capacidad hace referencia al tope físico que, en el caso de `a`, es de 50 valores de tipo `int`.

Consideremos ahora estas líneas de código:

```
int a[50];
a[0]=8;
a[1]=4;
a[2]=6;
```

En la siguiente figura, vemos representado el *array* `a` donde podemos observar que, si bien su capacidad permite contener hasta 50 valores enteros, su longitud es 3 porque actualmente solo tiene 3 elementos.

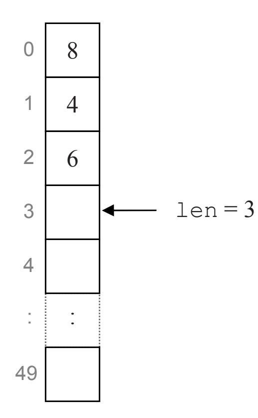


Fig. 7.6 *Array* con capacidad para 50 elementos y longitud actual (`len`) igual a 3.

Como en C los *arrays* se numeran desde cero resulta que en un *array* con longitud 3 (como el de la figura anterior) sus elementos ocuparán las posiciones comprendidas entre 0 y 2. Por lo tanto, `len` (su longitud) no solo indicará la cantidad de elementos contenidos en el *array* sino que además coincidirá con la primera posición libre del *array*.

7.4.2 Agregar un elemento al array

Esta operación permite agregar un elemento en la primera posición libre del *array*. Es decir, al final.

Si consideramos que el *array* está vacío entonces su longitud (`len`) será cero y este valor coincidirá con la posición en la que el elemento debe ser agregado. Luego de esto tendremos que incrementar `len` ya que la longitud del *array* pasará a ser 1.

Desarrollaremos la función `agregar` que tendrá el siguiente encabezado:

```
int agregar(int a[], int* len, int v)
```

donde `a` es el *array* en cuya primer posición libre asignaremos el valor `v`.

La función retornará la posición en la que se agregó a `v`, que siempre coincidirá con el valor de `len-1`, e incrementará el valor de `len` ya que al agregar un nuevo elemento al *array* su longitud debe ser incrementada. Veamos el diagrama de la función:

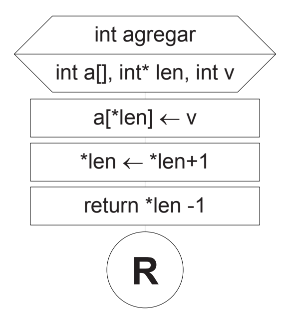


Fig. 7.7 Agrega un elemento al final del *array*.

Dado que la función agrega un elemento en el *array*, luego de esto su longitud habrá crecido. Esto queda reflejado al incrementar el valor de `len`. Por este motivo, `len` se recibe por referencia.

Veamos el código de la función y un programa que agrega valores enteros sobre un *array* y luego muestra su contenido.

arrays.c

```
int agregar(int a[],int* len,int v)
{
   a[*len]=v;
   *len=*len+1;
   return len-1;
}
```

testAgregar.c

```
#include <stdio.h>
#include "arrays.h"

int main()
{
   int arr[50];
   int len = 0; // la longitud inicial es cero

   agregar(arr,&len,3);
   agregar(arr,&len,5);
   agregar(arr,&len,7);
   agregar(arr,&len,9);

   for(int i=0; i<len; i++)
   {
      printf("%d\n",arr[i]);
   }

   return 0;
}
```

La función `agregar` incrementa el valor de `len` por lo tanto el programa principal (o quién invoque a la función) está eximido de realizar esta tarea. Luego de invocar 4 veces a la función el valor de `len` será 4 ya que el *array* tendrá asignados 4 elementos ubicados entre las posiciones 0 y 3.

El código de la función podría reducirse si lo escribimos de la siguiente manera:

```
int agregar(int a[],int* len,int v)
{
   a[(*len)++]=v;
   return *len-1;
}
```

7.4.3 Búsqueda secuencial

Esta operación consiste en recorrer secuencialmente el *array* para determinar si este contiene o no un determinado valor.

Desarrollaremos la función `buscar` que recibirá el *array*, su longitud y el elemento que queremos determinar si está o no contenido dentro del *array*. La función retornará la posición en la cual el *array* contiene al elemento que estamos buscando o un valor negativo si el elemento no está contenido dentro del *array*.

El encabezado de la función será el siguiente:

```
int buscar(int a[], int len, int v)
```

donde `a` es el *array*, `len` su longitud y `v` el elemento cuyo valor queremos determinar si está o no contenido dentro del *array* `a`.

La estrategia para desarrollar la función `buscar` será la siguiente: recorrer el *array* comenzando por la posición 0 y avanzando hacia la siguiente posición siempre y cuando no estemos excediendo su longitud y no encontremos lo que estamos buscando.

Veamos el diagrama:

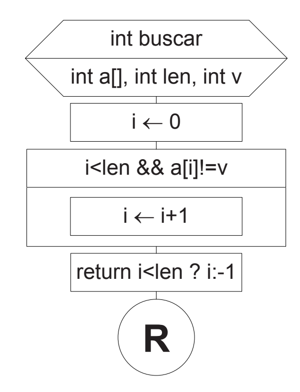


Fig. 7.8 Búsqueda secuencial sobre un *array*.

El ciclo itera mientras que `i<len` (es decir, que no excedamos la longitud del *array*) y mientras que `a[i]!=v` (mientras que no encontremos lo que buscamos).

Al finalizar el ciclo retornamos el valor de `i` o -1 según se verifique o no que `i<len`. Es decir que si no excedimos la longitud del *array* entonces el ciclo finalizó porque `a[i]` contiene a `v`, pero si `i` superó a `len` significa que `v` no está contenido en `a`.

A continuación, veremos la codificación de la función `buscar` y un programa en donde se invoca a la función para determinar si un *array* contiene o no un valor determinado.

arrays.c

```
//:
int buscar(int a[],int len,int v)
{
   int i=0;
   while( i<len && a[i]!=v )
   {
      i=i+1;
   }

   return i<len?i:-1;
}
```

■

testBuscar.c

```
#include <stdio.h>
#include "arrays.h"

int main()
{
   int arr[50];
   int len = 0;

   agregar(arr,&len,2);
   agregar(arr,&len,3);
   agregar(arr,&len,5);
   agregar(arr,&len,7);
   agregar(arr,&len,9);

   printf("2 esta en la posicion: %d\n",buscar(arr,len,2));
   printf("3 esta en la posicion: %d\n",buscar(arr,len,3));
   printf("5 esta en la posicion: %d\n",buscar(arr,len,5));
   printf("7 esta en la posicion: %d\n",buscar(arr,len,7));
   printf("9 esta en la posicion: %d\n",buscar(arr,len,9));
   printf("2 esta en la posicion: %d\n",buscar(arr,len,2));

   return 0;
}
```

La función `buscar` puede reescribirse de la siguiente manera:

```
int buscar(int a[],int len,int v)
{
   int i;
   for(i=0;i<len && a[i]!=v;i++);
   return i<len?i:-1;
}
```

7.4.4 Buscar y agregar

Esta operación es una combinación de las dos anteriores. La idea es buscar un elemento en un *array* y retornar la posición donde se encontró o bien agregarlo al final en caso de no haberlo encontrado.

La función `buscarYAgregar` tendrá el siguiente encabezado:

```
int buscarYAgregar(int a[], int* len, int v, int *enc)
```

Asignará *true* o *false* en el parámetro `enc` según el elemento `v` se encuentre previamente o no en el *array* `a` y su valor de retorno representará:

- si `enc` es *true*: la posición en donde `a` previamente contenía a `v`.
- si `enc` es *false*: la posición en donde (ahora) `a` contiene a `v`.

El diagrama de la función es el siguiente:

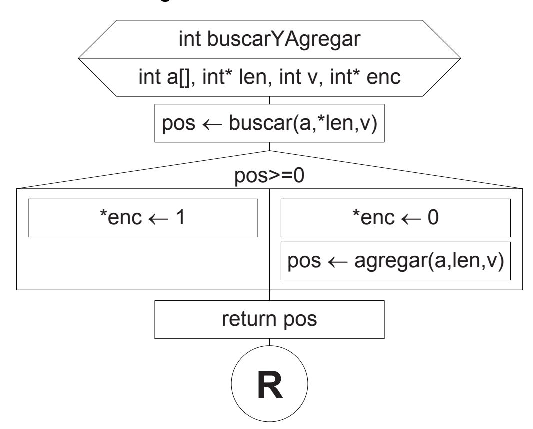


Fig. 7.9 Busca un elemento en un *array* y si no lo encuentra lo agrega.

Veamos su codificación:

arrays.c

```
//:
int buscarYAgregar(int a[], int* len, int v, int* enc)
{
   int pos=buscar(a,*len,v);

   if( pos>=0 )
   {
      *enc = 1;
   }
   else
   {
      *enc=0;
      pos=agregar(a,len,v);
   }

   return pos;
}
```

Con lo estudiado hasta aquí podemos resolver la segunda versión del problema 7.1 en la que el conjunto de números cual ingresará el usuario no está acotado.

Problema 7.3

Dado un conjunto de números enteros determinar cuántas veces se repite cada uno. Los datos se ingresan sin ningún orden y finalizan al llegar el valor 0. Se garantiza que a lo sumo habrá 100 números diferentes.

Análisis

Veamos un lote de datos acorde al enunciado del problema.

`A = {4, 521, -7, 912, 1045, 1, 521, 2, 1, 3, -7, -43}`

La principal diferencia entre este problema y el problema 7.1 es que aquí los datos no están acotados; por lo tanto, no podemos usarlos como índice de acceso directo a un *array* de contadores.

Sabemos que a lo sumo habrá 100 números distintos; por lo tanto, necesitaremos disponer de 100 contadores, pero la relación entre el contador y el número del conjunto será indirecta.

La estrategia para resolver el problema será la siguiente: utilizaremos dos *arrays*: `aNum` y `aCont`. A medida que leamos cada número lo “buscaremos y agregaremos” en `aNum`. Esto nos garantizará agregar el número al *array* una única vez y así tener una relación unívoca entre su valor y la posición en la que fue agregado. Esta posición la utilizaremos para incrementar el contador implementado en el *array* `aCont`. Como vemos, ahora el acceso a `aCont` es indirecto, previa búsqueda en `aNum`.

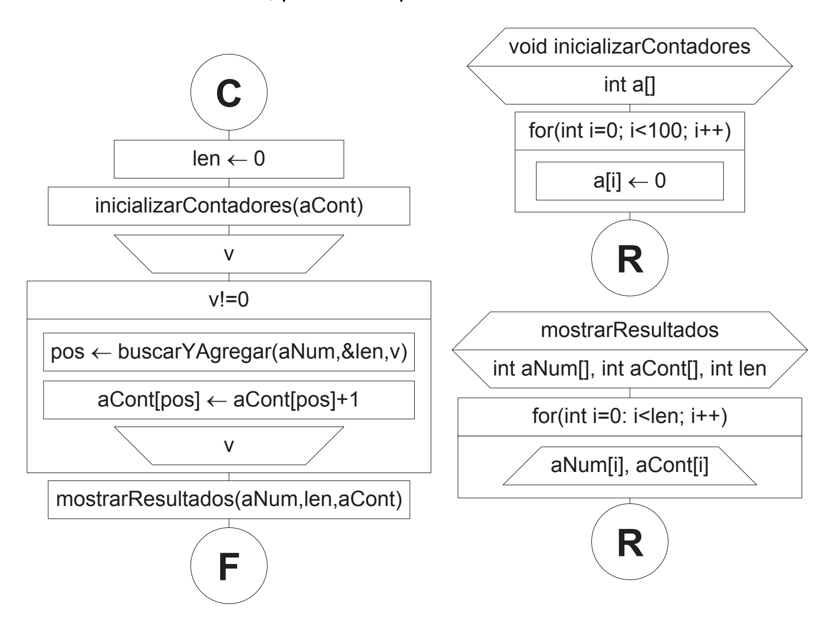
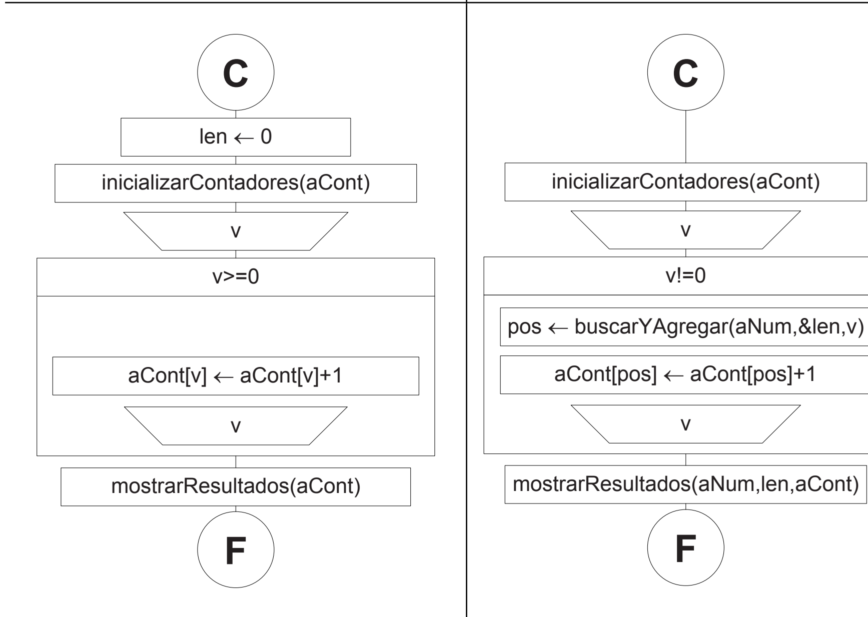


Fig. 7.10 Resolución del problema

Comparemos los programas principales de los problemas 7.1 y 7.3.

Solución del problema 7.1 Acceso directo a un *array* de contadores	**Solución del problema 7.3** Acceso indirecto a un *array* de contadores



En este caso usamos el valor de `v` como índice para acceder al *array* `aCont` e incrementar el contador.	En este problema `aNum` es un "*array* nomenclador" en el cual almacenamos los diferentes valores de `v` y sobre el que hacemos una búsqueda para obtener la posición del elemento cuyo contador queremos incrementar.

Fig. 7.11 Comparación entre acceso directo y acceso indirecto a un *array*.

Veamos la codificación del problema 7.3.

`problema7.3.h`

```
void inicializarContadores(int[]);
void mostrarResultados(int[], int, int[]);
```

`problema7.3p.c`

```
#include <stdio.h>
#include "arrays.h"

int main()
{
   int aNum[100],aCont[100];
   int len=0;

   inicializarContadores(aCont);
```

```
   int v;
   printf("Ingrese un valor: ");
   scanf("%d",&v);

   int enc,pos;

   while( v!=0 )
   {
      pos = buscarYAgregar(aNum,&len,v,&enc);
      aCont[pos]=aCont[pos]+1;

      printf("Ingrese un valor: ");
      scanf("%d",&v);
   }

   mostrarResultados(aNum,len,aCont);

   return 0;
}
```

problema7.3f.c

```
#include <stdio.h>

void inicializarContadores(int a[])
{
   for(int i=0; i<100; i++)
   {
      a[i]=0;
   }
}

void mostrarResultados(int aNum[],int len,int aCont[])
{
   for(int i=0; i<len; i++)
   {
      printf("%d paracece %d veces\n",aNum[i],aCont[i]);
   }
}
```

7.4.5 Insertar un elemento

A través de esta operación, podremos insertar un elemento en una determinada posición del *array* siempre y cuando esta posición esté comprendida entre 0 y `len`.

Llamemos `pos` a la posición en la que vamos a insertar el elemento `v`.

El algoritmo consiste en desplazar hacia abajo a todos aquellos elementos comprendidos entre `pos` y `len-1` para dejar libre la posición `pos` y poder asignar allí a `v`.

En el siguiente gráfico, insertamos el valor 5 en la posición 2 de un *array*:

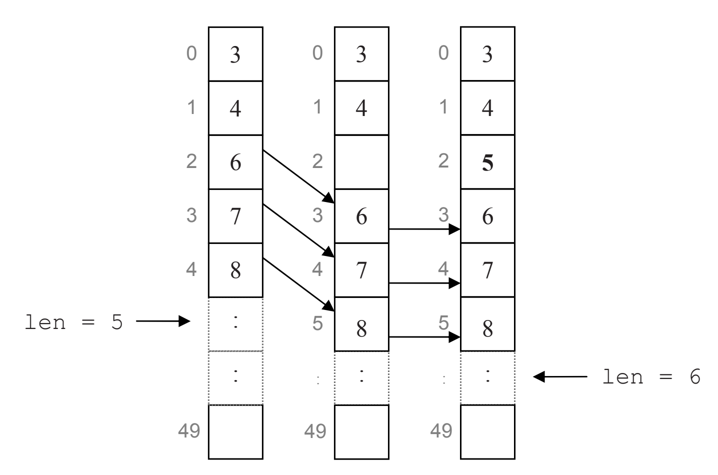


Fig. 7.12 Desplazamiento e inserción en un *array*.

Debemos tener en cuenta que el desplazamiento debe hacerse desde abajo hacia arriba ya que de lo contrario perderemos la información contenida en el *array*.

Pensemos entonces en desarrollar la función `insertar` cuya cabecera será la siguiente:

```
void insertar(int a[], int* len, int v, int pos)
```

Veamos el diagrama:

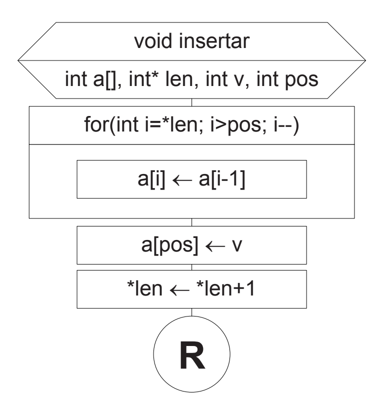


Fig. 7.13 Inserta un elemento en una determinada posición del *array*.

Para desplazar los elementos "de abajo hacia arriba", utilizamos un `for` dentro del cual `i` va decrementándose entre `len` y `pos+1`. Luego asignamos en `a[i]` el valor de `a[i-1]`. Esto hará que, en la primer iteración, asignemos en `a[len]` el elemento contenido en `a[len-1]`. En segunda asignaremos en `a[len-1]` el elemento ubicado en `a[len-2]` y así sucesivamente hasta que en la última iteración asignaremos en `a[pos+1]` el elemento ubicado en `a[pos]` dejando esta posición libre para poder asignar el valor `v`.

A continuación, veremos la codificación de la función `insertar` y un programa que inserta 4 valores en un *array*, todos en la posición cero y luego muestra su contenido.

arrays.c

```c
// :
void insertar(int a[],int* len,int v,int pos)
{
   for(int i=*len; i>pos; i--)
   {
      a[i]=a[i-1];
   }

   a[pos]=v;
   *len=*len+1;
}
```

testInsertar.c

```c
#include <stdio.h>
#include "arrays.h"

int main()
{
   int n;
   int arr[50];
   int len=0;

   insertar(arr,&len,4,0);
   insertar(arr,&len,3,0);
   insertar(arr,&len,2,0);
   insertar(arr,&len,1,0);

   for(int i=0; i<len; i++)
   {
      printf("%d\n",arr[i]);
   }

   return 0;
}
```

La salida de este programa será:

```
1
2
3
4
```

7.4.6 Eliminar un elemento

Esta es la operación inversa a la de insertar un elemento. La idea es eliminar del *array* al elemento que se encuentre en una posición especificada. El algoritmo consistirá en desplazar hacia arriba a todos aquellos elementos ubicados entre `pos+1` y `len` siendo `len` la longitud del *array* y `pos` la posición del elemento que queremos eliminar. También será necesario decrementar el valor de `len` porque, luego de eliminar un elemento, la longitud del *array* habrá disminuido.

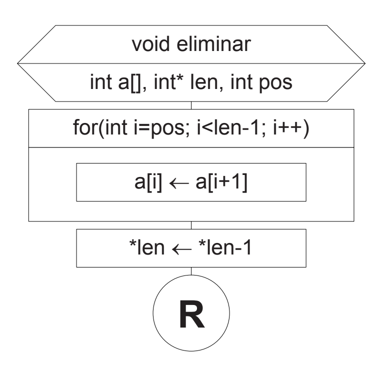


Fig. 7.14 Elimina el elemento ubicado en una posición especificada.

La codificación de la función `eliminar` más un programa que genera un *array* y luego elimina uno de sus elementos es la siguiente:

arrays.c

```
// :
void eliminar(int a[], int* len, int pos)
{
   for(int i=pos; i<*len-1; i++)
   {
      a[i]=a[i+1];
   }

   *len=*len-1;
}
```

testEliminar.c

```
#include<stdio.h>
#include "arrays.c"

int main()
{
   int arr[50];
   int len = 0;

   agregar(arr,&len,0);
   agregar(arr,&len,1);
   agregar(arr,&len,2);
   agregar(arr,&len,3);
   agregar(arr,&len,4);
   agregar(arr,&len,5);

   eliminar(arr,&len,3);

   for(int i=0; i<len; i++)
   {
      printf("%d\n",arr[i]);
   }

   return 0;
}
```

La salida de este programa será:

```
0
1
2
4
5
```

7.4.7 Insertar en orden

Esta operación permite insertar un valor `v` en un *array* respetando el criterio de ordenamiento que mantienen sus elementos. Es decir que el contenido del *array* debe estar ordenado.

El algoritmo consiste en recorrer el *array* comenzando desde la posición 0 y avanzando mientras que no excedamos su longitud y mientras que el elemento que encontramos en la *i-ésima* posición sea menor o igual al que buscamos. Luego lo insertaremos en la posición `i`.

La función `insertarEnOrden` tendrá el siguiente encabezado:

```
int insertarEnOrden(int a[], int* len, int v, int* encontro)
```

Veamos el diagrama y luego lo analizaremos con más detalle.

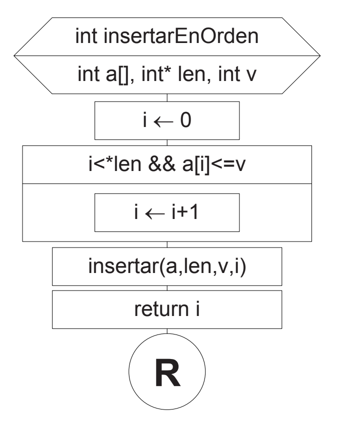


Fig. 7.15 Inserta un elemento en el *array* respetando el criterio de ordenamiento.

Supongamos que el *array* `a` es el siguiente:

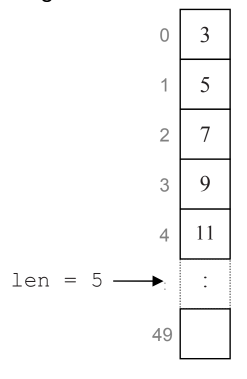


Fig. 7.16 *Array* de enteros ordenado.

Sea `i` una posición del *array* y `a[i]` el elemento contenido en esta posición. Al valor que vamos a insertar lo llamaremos `v` y, por supuesto, llamaremos `len` a la longitud del *array*. Los casos que tenemos que estudiar para garantizar que el algoritmo funcione correctamente son los siguientes:

- Caso 1 - `v>=`*m* y `v<=`*M* donde *m* y *M* son, respectivamente, el mínimo y el máximo valor del *array*. Además, como el *array* está ordenado, se verifica que *m* está en la posición 0 y *M* está en la posición `len-1`.
- Caso 2 - `v>`*M*. Vamos a insertar un valor más grande que el mayor.
- Caso 3 - `v<`*m*. Vamos a insertar un valor más chico que el menor.

Caso 1 - Supongamos que `v` vale 8 entonces si recorremos el *array* mientras `i<len` y mientras que `a[i]<=v` llegaremos hasta la posición `i=3` dado que `a[i]` es el primer elemento mayor que `v`. Justamente, en `i=3` es donde deberíamos insertar el 8 para que el *array* siga manteniéndose ordenado.

Caso 2 - Supongamos que `v` vale 13. Si recorremos el *array* mientras `i<len` y mientras que `a[i]<=v` llegaremos hasta la posición `i=5` ya que este valor de `i` supera la posición del último elemento del *array* y, justamente, es en `i=5` donde deberíamos insertar a `v` para que el *array* siga estando ordenado.

Caso 3 - Supongamos ahora que `v` vale 2. Si recorremos el *array* mientras `i<len` y mientras que `a[i]<=v` llegaremos hasta la posición `i=0` ya en esta posición se encuentra el primer elemento mayor que `v` es allí donde deberíamos insertar a 2.
Dado que en todos los casos `i` coincide con la posición en donde debemos insertar a `v` invocamos a la función `insertar` analizada más arriba para insertar a `v` en la posición `i` del *array* `a`.

La función `insertarEnOrden` retorna la posición en la que se insertó a `v` que, como ya sabemos, coincide con el valor de `i`.
Veamos ahora la codificación de `insertarEnOrden` y un programa que la utiliza para insertar elementos en un *array* que luego lo recorre para verificar que todos los elementos quedaron ordenados.

arrays.c

```
//:
int insertarEnOrden(int a[], int* len, int v)
{
   int i=0;
   while(i<*len && a[i]<=v)
   {
      i=i+1;
   }

   insertar(a,len,v,i);
   return i;
}
```
■

testInsertarEnOrden.c

```
#include <stdio.h>
#include "arrays.h"

int main()
{
   int arr[50];
   int len = 0;
```

```
   insertarEnOrden(arr,&len,4);
   insertarEnOrden(arr,&len,0);
   insertarEnOrden(arr,&len,1);
   insertarEnOrden(arr,&len,3);
   insertarEnOrden(arr,&len,5);
   insertarEnOrden(arr,&len,2);

   for(int i=0; i<len; i++)
   {
      printf("%d\n",arr[i]);
   }

   return 0;
}
```

La salida de este programa será:

```
0
1
2
3
4
5
```

7.4.8 Buscar en orden

Esta operación realiza la búsqueda secuencial de un elemento `v` sobre un *array* `a` considerando que el contenido de `a` está ordenado. Con esta premisa no será necesario recorrer el *array* hasta el final para determinar si contiene o no a `v` ya que si durante la recorrida encontramos que `a[i]` es mayor que `v` será porque `a` no contiene a `v`.

Desarrollaremos la función `buscarEnOrden` con el siguiente encabezado:

```
int buscarEnOrden(int a[], int len, int v, int* enc)
```

La función debe buscar a `v` en `a`. Si lo encuentra retornará la posición en la que `a` contiene a `v` y asignará *true* al parámetro `enc`. Si no asignará *false* a `enc` y retornará la posición en la que `a` debería contener a `v` para que, si lo fuéramos a insertar, el *array* continúe manteniendo los elementos en orden. Veamos el algoritmo:

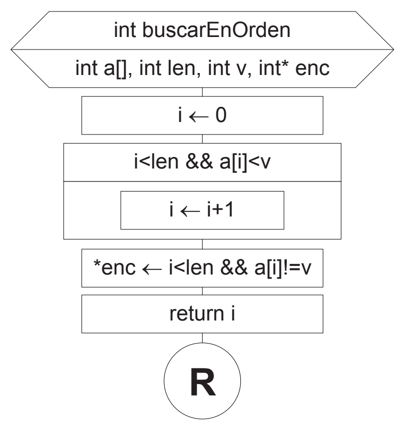


Fig. 7.17 Busca un elemento en un *array* ordenado.

La codificación es la siguiente:

arrays.c

```
//:
int buscarEnOrden(int a[], int len, int v, int* enc)
{
   int i=0;
   while(i<len && a[i]<v)
   {
      i=i+1;
   }

   *enc=i<len;
   return i;
}
```

7.4.9 Buscar e insertar en orden

Esta operación combina las dos operaciones anteriores y permite insertar (en orden) un valor que no esté contenido en el *array*. Si intentamos insertar un valor repetido retornará la posición en la cual el *array* contiene a dicho valor.

El encabezado será el siguiente:

```
int buscarEInsertarEnOrden(int a[], int* len, int v, int* enc)
```

La función asignará en `enc` *true* o *false* según el valor `v` se encuentre o no en el *array* `a` y retornará un valor entero cuyo significado será:

- si `enc` es *true*, la posición donde `a` contiene a `v`.
- si `enc` es *false*, la posición de `a` donde se insertó a `v`.

Teniendo resueltas las funciones `buscarEnOrden` e `insertarEnOrden`, el algoritmo que resuelve esta operación es trivial y se limita a "buscar en orden" a `v` en `a` y luego a "insertarlo en orden" en caso de no haberlo encontrado.

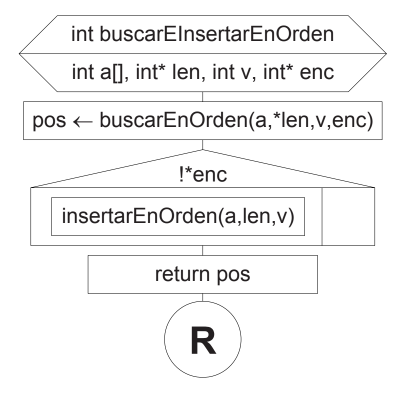


Fig. 7.18 Busca un elemento y si no lo encuentra lo inserta manteniendo el orden del *array*.

Veamos el código de la función:

arrays.c

```
//:
int buscarEInsertarEnOrden(int a[], int* len, int v, int* enc)
{
   int pos=buscarEnOrden(a,*len,v,enc);

   if(!*enc)
   {
      insertarEnOrden(a,len,v);
   }

   return pos;
}
```



Algoritmo de la burbuja

7.4.10 Ordenar arrays (algoritmo de la "burbuja")

Existen diferentes algoritmos a través de los cuales podemos ordenar los elementos contenidos en un *array*. En este capítulo, estudiaremos el algoritmo más simple (y también el menos eficiente) conocido como el "algoritmo de la burbuja". Los algoritmos más complejos y eficientes los analizaremos en los capítulos posteriores.

El algoritmo de la "burbuja" consiste en recorrer el *array* analizando la relación de precedencia que existe entre cada elemento y el elemento que le sigue para determinar si estos se encuentran ordenados entre sí y, en caso de ser necesario, permutarlos para que queden ordenados.

Genéricamente hablando, si `a` es el *array* que vamos a ordenar e `i` es un valor comprendido entre 0 y `len-1` entonces: si `a[i]>a[i+1]` significa que estos dos elementos se encuentran desordenados entre si y habrá que permutarlos.

En el siguiente gráfico, tomamos de a pares a los elementos del *array*, los comparamos y si corresponde los permutamos para que cada par de elementos quede ordenado entre sí. Primero evaluamos `a[0]` y `a[1]`, luego `a[1]` y `a[2]`, luego `a[2]` y `a[3]` y por último, `a[3]` y `a[4]`. El *array* tiene 4 elementos.

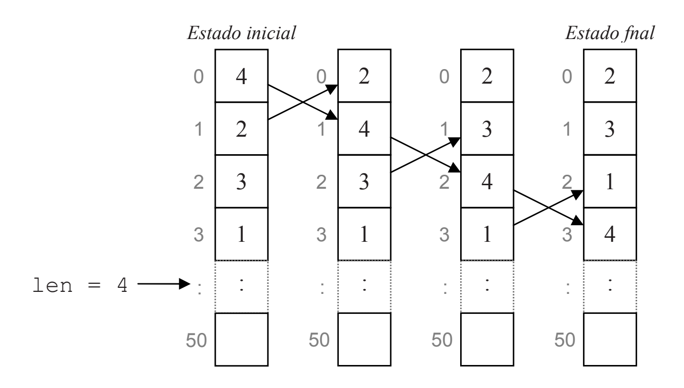


Fig. 7.19 Primera pasada de ordenamiento "burbuja".

Luego de realizar todas las comparaciones y permutaciones, el *array* quedó "un poco más ordenado" respecto de su estado inicial. Si volvemos a repetir la operación

seguramente el *array* quedará mejor aún (notemos que el estado inicial de esta nueva iteración coincide con el estado final de la iteración anterior). Veamos:

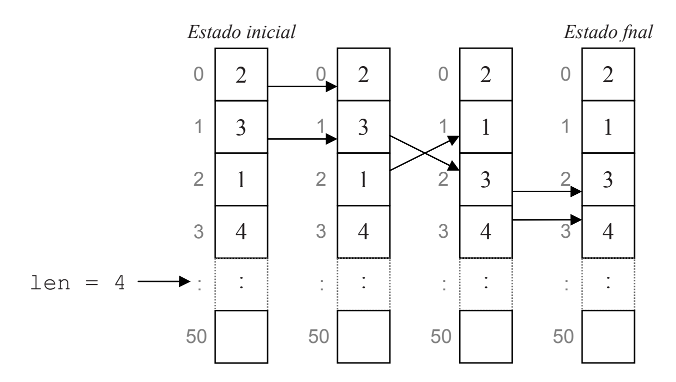


Fig. 7.20 Segunda pasada de ordenamiento "burbuja".

Efectivamente, el *array* quedó mucho mejor y a simple vista observamos que basta con una nueva iteración para realizar la última permutación con la cual su contenido quedará ordenado.

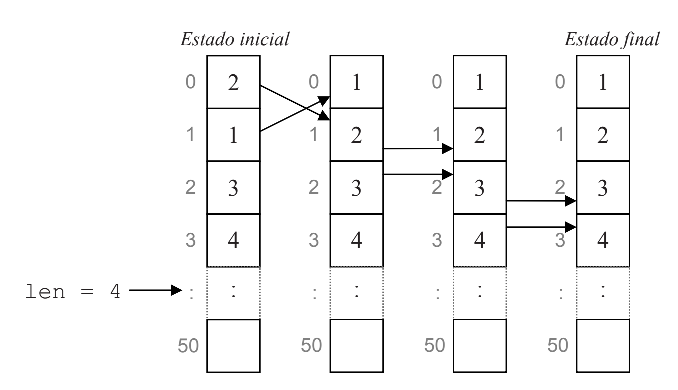


Fig. 7.21 Tercera pasada de ordenamiento "burbuja".

Desarrollaremos la función `ordenar` con el siguiente encabezado:

```
void ordenar(int a[], int len)
```

El algoritmo entonces será recorrer el *array* comparando `a[i]` con `a[i+1]` para permutarlos si no están en orden. El proceso finalizará cuando realicemos una iteración en la cual no haya sido necesario realizar ninguna permutación.

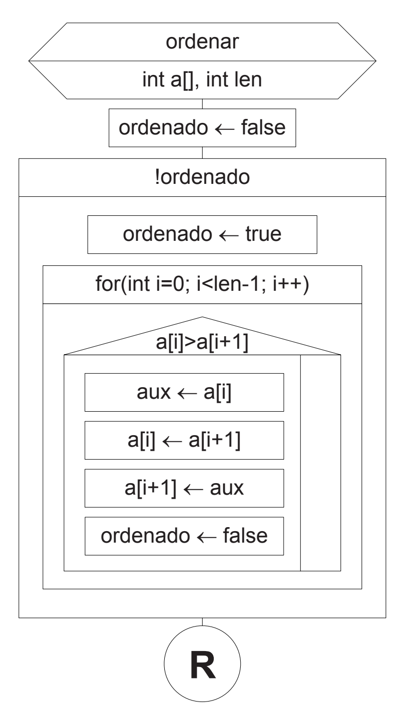


Fig. 7.22 Algoritmo "burbuja" para ordenar *arrays*.

Veamos la codificación de la función `ordenar` y un programa en el que definimos un *array* con 5 elementos, desordenado y lo ordenamos invocando a la función.

arrays.c

```
//:
void ordenar(int a[],int len)
{
   int ordenado=0;
   while( !ordenado )
   {
      ordenado=1;
      for(int i=0; i<len-1; i++)
      {
         if( a[i]>a[i+1] )
         {
            int aux=a[i];
            a[i]=a[i+1];
            a[i+1]=aux;
            ordenado=0;
         }
      }
   }
}
```

testOrdenar.c

```
#include <stdio.h>
#include "arrays.h"

int main()
{
   int a[]={5,4,3,2,1};
   int len=5;

   ordenar(a,len);

   for(int i=0; i<len; i++)
   {
      printf("%d\n",a[i]);
   }

   return 0;
}
```

7.4.11 Búsqueda binaria o dicotómica



Algoritmo de la búsqueda binaria

La búsqueda binaria es la misma búsqueda que llevamos a cabo cuando utilizamos un directorio telefónico para encontrar el número de teléfono de una persona.

Supongamos que el apellido de la persona que estamos buscando es “Gomez” entonces abrimos la guía, más o menos por la mitad, y analizamos si en esa página se detallan los apellidos que comienzan con la letra “G”. Si la página en la que abrimos contiene apellidos que comienzan con la letra “P” tendremos la pauta de que “Gomez” debe estar ubicado en las páginas anteriores ya que, como la guía está ordenada alfabéticamente, los apellidos que comienzan con “G” estarán detallados en alguna página anterior a la página que detalla apellidos que comienzan con “P”.

Ahora bien, si abrimos la guía en alguna página anterior y notamos que allí se detallan los apellidos que comienzan con la letra “B” podremos asegurar que el teléfono de “Gomez” estará en alguna página posterior a esta, pero anterior a la que inspeccionamos al principio.

El algoritmo de la búsqueda dicotómica consiste en inspeccionar el *array* descartando todos aquellos elementos que son superiores e inferiores al valor que estamos buscando. Dado que el *array* debe estar ordenado entonces los elementos superiores ocuparán posiciones posteriores y los elementos inferiores ocuparán posiciones anteriores a la del elemento que buscamos.

7.4.11.1 Implementación

Para implementar la búsqueda binaria de un valor `v` sobre un *array* `a` con longitud `len` utilizaremos dos variables: `i` y `j`. La primera será inicializada en cero (la posición del primer elemento del *array*) y la segunda será inicializada en `len-1` (la posición del último elemento). Luego calcularemos la posición intermedia: `k=(i+j)/2`. Recordemos que la división entre valores enteros devuelve un valor entero.

El siguiente paso será preguntar si `a[k]` es igual a `v` (el valor que buscamos). Esto es: “¿Encontré lo que buscaba?”. Si se verifica la igualdad anterior tendremos resuelta

la búsqueda, pero si no, y resulta que `a[k]` no es igual a `v` entonces tendrá que ser mayor o menor.

- Si `a[k]>v` entonces `v` debería estar entre las posiciones 0 y `k-1` de `a`.
- Si `a[k]<v` entonces `v` debería estar entre las posiciones `k+1` y `len-1`.

Según lo anterior, ajustaremos los valores de `i` y `j` de la siguiente manera:

- Si `a[k]>v` entonces asigno `j=k-1`.
- Si `a[k]<v` entonces asigno `i=k+1`.

Luego de estas asignaciones habremos descartado la mitad anterior o la mitad posterior a la posición del *array* indicada por `k` y podremos repetir el proceso.

En el siguiente gráfico tenemos un *array* ordenado y queremos determinar si contiene (por ejemplo) el valor 5.

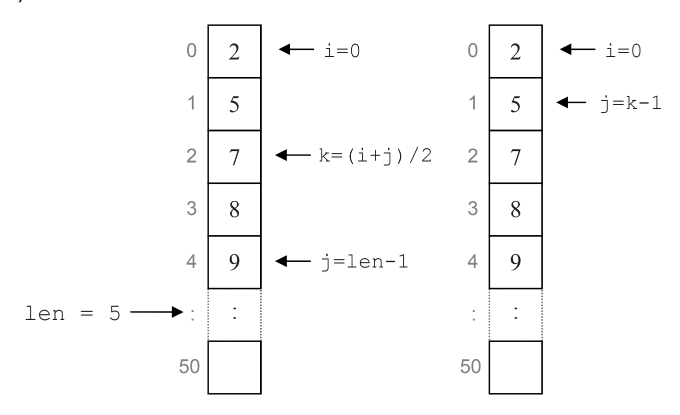


Fig. 7.23 Búsqueda binaria, primer intento.

Vemos que inicializamos `i=0`, `j=len-1` y calculamos el promedio `k=(i+j)/2`. Como en esta posición encontramos el valor 7 y resulta que 7 es mayor que 5 entonces descartamos todos aquellos elementos del *array* ubicados en posiciones posteriores a `k` asignando a `j=k-1`.

Si repetimos la operación considerando ahora el nuevo valor de `j` tendremos lo siguiente:

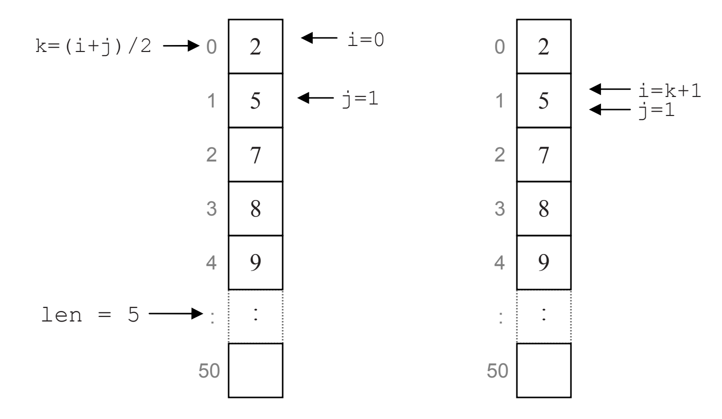


Fig. 7.24 Búsqueda binaria, segundo intento.

Como encontramos el valor 2, pero buscamos a 5, de estar, estará en alguna de las posiciones ubicadas entre `k+1` y `j`. Por lo tanto, asignamos `i=k+1`.

Si repetimos la operación una vez más resultará que `k=(i+j)/2` será 1, posición en la cual el *array* contiene el valor que buscamos.

Si en lugar de buscar el valor 5 estuviéramos buscando el valor 4 resultará que `a[k]` contiene a 5, pero nosotros buscamos a 4 que, de estar, estará ubicado entre las posiciones `i` y `k-1`. Luego asignamos `j=k-1` y obtenemos que `j` es menor que `i`, lo que nos estará indicando que el *array* no contiene el elemento que buscamos.

Con esta base desarrollaremos la primera versión de la función `busquedaBinaria` cuyo encabezado será el siguiente:

```
int busquedaBinaria(int a[], int len, int v, int* enc)
```

La función permite buscar el valor `v` dentro del *array* `a` cuya longitud es `len`. Si `a` contiene a `v` entonces retornará la posición de `v` dentro de `a` y asignará *true* a `enc`. De lo contrario, simplemente asignará *false* en dicho parámetro.

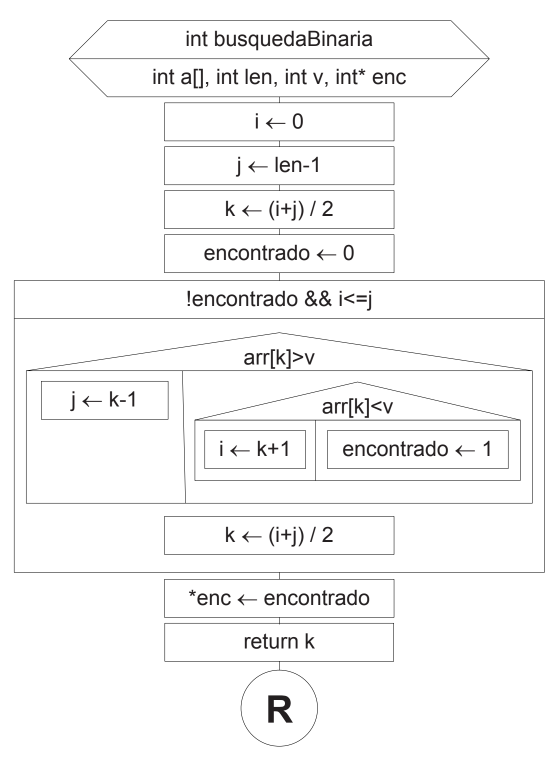


Fig. 7.25 Algoritmo de la búsqueda binaria

Veamos la codificación y luego un programa en donde definimos un *array* con valores enteros, ordenado ascendentemente e invocamos varias veces a la función para analizar sus resultados.

arrays.c

```
//:
int busquedaBinaria(int a[], int len, int v, int* enc)
{
   int i=0;
   int j=len-1;
   int k=(i+j)/2;

   int encontrado=0;

   while( !encontrado && i<=j )
   {
      if( a[k]>v )
      {
         j=k-1;
      }
      else
      {
         if( a[k]<v )
         {
            i=k+1;
         }
         else
         {
            encontrado=1;
         }
      }

      k=(i+j)/2;
   }

   *enc=encontrado;
   return k;
}
```

Veamos ahora el programa que prueba la función anterior:

testBusquedaBinaria.c

```
#include <stdio.h>
#include "arrays.h"

int main()
{
   // defino un array de enteros y ordenado
   int arr[50]={1,4,7,9,10,12};
   int len=6;

   int v,pos,enc;

   for(int i=-3; i<15; i++)
   {
      pos = busquedaBinaria(arr,len,i,&enc);

      printf("%d",i);
      if( enc )
      {
         printf(" [encontrado], ");
```

```
      }
      else
      {
         printf(" [NO encontrado], ");
      }
      printf("pos=%d\n",pos);
   }
   return 0;
}
```

La salida del programa es la siguiente:

```
-3 [NO encontrado], pos=0
-2 [NO encontrado], pos=0
-1 [NO encontrado], pos=0
0 [NO encontrado], pos=0
1 [encontrado], pos=0
2 [NO encontrado], pos=0
3 [NO encontrado], pos=0
4 [encontrado], pos=1
5 [NO encontrado], pos=1
6 [NO encontrado], pos=1
7 [encontrado], pos=2
8 [NO encontrado], pos=2
9 [encontrado], pos=3
10 [encontrado], pos=4
11 [NO encontrado], pos=4
12 [encontrado], pos=5
13 [NO encontrado], pos=5
14 [NO encontrado], pos=5
```

Ahora analizaremos estos resultados para encontrar la relación existente entre el valor de retorno de la función `busquedaBinaria` y aquellos valores que no han sido encontrados dentro del *array*. Para facilitar este análisis, reorganizaremos la salida del programa en dos columnas y desecharemos aquellos resultados en los que `enc` resulto verdadero. Recordemos los datos contenidos en el *array*:

```
int arr[]={1,4,7,9,10,12};
```

<table>
<tr><th>Valor a buscar</th><th>Valor de retorno</th><th></th></tr>
<tr><td>-3</td><td>0</td><td rowspan="4">Todos estos valores son menores que el menor valor contenido en el array.
El valor de retorno coincide con la posición en la que podrían ser insertados sin que esto implique alterar el orden del array.</td></tr>
<tr><td>-2</td><td>0</td></tr>
<tr><td>-1</td><td>0</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>2</td><td>0</td><td rowspan="8">Todos estos son valores mayores que el menor de los valores del array. Si sumamos 1 al valor de retorno que devuelve la función tendremos la posición en la que todos estos podrían insertarse en el array sin alterar su ordenamiento.</td></tr>
<tr><td>3</td><td>0</td></tr>
<tr><td>5</td><td>1</td></tr>
<tr><td>6</td><td>1</td></tr>
<tr><td>8</td><td>2</td></tr>
<tr><td>11</td><td>4</td></tr>
<tr><td>13</td><td>5</td></tr>
<tr><td>14</td><td>5</td></tr>
</table>

Según la tabla anterior, cuando `enc` es *false* y `v<a[k]`, observamos que `k` coincide con la posición que `v` debería ocupar dentro de `a` si lo fuéramos a insertar.

En cambio, si `enc` es *false* y `v>a[k]` entonces la posición que le correspondería ocupar a `v` dentro de `a` sería `k+1`.

Aplicando el resultado de este análisis a la función `busquedaBinaria` tendremos la segunda versión de la función modificando la línea del `return`.

```
return v<=a[k]?k:k+1;
```

A continuación, modificaremos el programa anterior para complementar el *array* `arr` con todos los valores consecutivos comprendidos entre -3 y 14.

```
int main()
{
   int arr[50]={1,4,7,9,10,12};
   int len=6;

   int v,pos,enc;

   for(int i=-3; i<15; i++)
   {
      pos = busquedaBinaria(arr,len,i,&enc);

      if( !enc )
      {
         insertar(arr,&len,i,pos);
      }
   }

   // muestro como quedo el array
   for(int i=0; i<len; i++)
   {
      printf("%d\n",arr[i]);
   }

   return 0;
}
```

7.5 Arrays multidimensionales

Sabemos que un *array* representa un conjunto de variables del mismo tipo de datos. Cuando el tipo de datos del *array* es en sí mismo un *array* hablamos de *arrays* multidimensionales.

En el lenguaje de programación Pascal, esto se observa claramente al momento de declarar una variable de tipo *array*. Veamos:

```
// define un array de enteros
var a:array[1..3] of integer;

// define un array de arrays de enteros
var b:array[1..3] of array[1..5] of integer;

// define un array de arrays de arrays de enteros
var c:array[1..3] of array[1..5] of array[1..4] of integer;
```

7.5.1 Arrays bidimensionales (matrices)

Un *array* bidimensional es aquel cuyo tipo de datos es, en sí mismo, otro *array*. A un *array* con estas características lo llamamos “matriz” ya que gráficamente lo representamos como un conjunto de filas y columnas. Las celdas que se forman en las intersecciones de estas son los elementos del *array* bidimensional.

En C definimos una matriz de enteros de la siguiente forma:

```
int m[4][3];
```

Luego de esto la variable `m` representa un *array* de 4 elementos de tipo `int[3]`. Esto es: un *array* con capacidad para contener 4 *arrays*, cada uno con capacidad para contener 3 enteros. O bien: una matriz de 4 filas y 3 columnas de enteros.

Gráficamente, lo podemos representar de cualquiera de las siguientes formas:

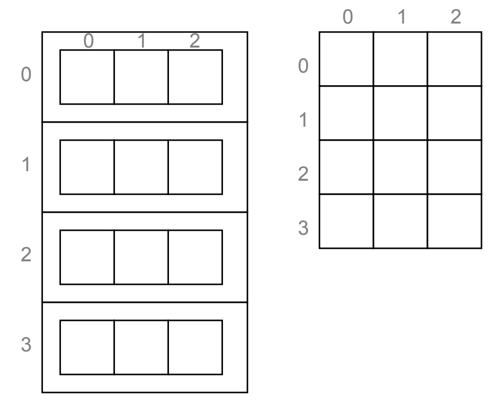


Fig. 7.26 Representación gráfica de un *array* bidimensional o matriz.

Luego, para asignar un valor en alguna de las celdas de la matriz debemos especificar la fila y la columna como vemos a continuación:

```
// asigno el valor 5 en la celda ubicada en la fila 1, columna 2
m[1][2]=5;
```

En el siguiente código, definimos una matriz de 4 filas y 3 columnas e inicializamos todas sus celdas con el valor 0.

```
int m[4][3];
for(int i=0; i<4; i++)
{
   for(int j=0; j<3; j++)
   {
      m[i][j]=0;
   }
}
```

En C también podemos inicializar una matriz en el mismo momento en que la estamos declarando, veamos:

```
int m[][] = { {1,2,3}, {4,5,6}, {7,8,9}, {10,11,12} };
```

Esto dimensiona a la matriz `m` con 4 filas y 3 columnas y además, asigna cada uno de los valores especificados en la celda correspondiente.

7.5.1.1 Recorrer una matriz por fila/columna

En el siguiente código, imprimimos cada una de las celdas de la matriz declarada más arriba recorriéndola por sus filas y luego por sus columnas.

```
for(int i=0; i<4; i++)
{
   for(int j=0; j<3; j++)
   {
      printf("%d ",m[i][j]);
   }

   printf("\n");
}
```

La salida será:

```
 1  2  3
 4  5  6
 7  8  9
10 11 12
```

7.5.1.2 Recorrer una matriz por columna/fila

En el siguiente código, imprimimos cada una de las celdas de la matriz recorriéndola por sus columnas y luego por sus filas.

```
for(int j=0; i<3; j++)
{
   for(int i=0; i<4; i++)
   {
      printf("%d ",m[i][j]);
   }

   printf("\n");
}
```

La salida será:

```
 1  4  7  10
 2  5  8  11
 3  6  9  12
```

Problema 7.4

Se ingresa un conjunto de ternas de valores que representan el año, el grado y la cantidad de alumnos que se inscribieron en un colegio durante ese año y para ese grado en particular. Solo se ingresará la información comprendida entre los años 2000 y 2009. En el colegio, los alumnos cursan desde el primer grado y hasta el séptimo. Los datos se ingresan sin ningún tipo de orden.

Se pide:

1. Emitir un listado ordenado por año detallando para cada año la cantidad de inscritos por grado.
2. Emitir un listado ordenado por grado detallando para cada grado la cantidad de inscritos por año.

Análisis

Para emitir los listados solo necesitamos ordenar los datos de entrada: primero por año/grado y luego por grado/año. Para esto, utilizaremos una matriz de 10 filas por 7 colum-

nas donde cada fila representará un año y cada columna un grado. Luego, cada celda contendrá la cantidad de inscritos registrados durante ese año (fila), para ese grado (columna). Es decir, utilizaremos el año y el grado como índices para acceder directamente a las celdas de la matriz y asignar en cada una la cantidad de inscritos correspondiente.

Luego, si recorremos la matriz por filas y luego por columnas veremos los datos ordenados por año y luego por grado. En cambio, si la recorremos por columna y luego por fila los veremos ordenados por grado y luego por año.

Desarrollemos el programa principal:

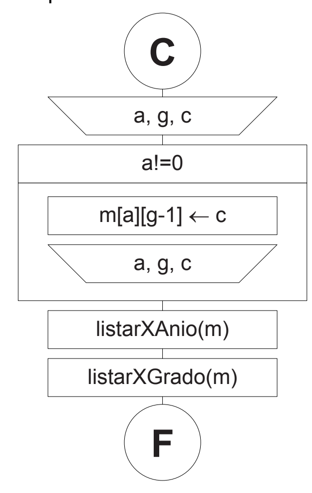


Fig. 7.27 Programa principal, carga los datos en una matriz.

Como comentamos más arriba, utilizamos el año `a` y el grado `g` para acceder a la celda de la matriz `m` y asignar allí la cantidad `c` de inscritos.

La fila 0 representa al año 2000, la fila 1 representa al año 2001, etc. Dado que los grados comienzan desde 1, el acceso a las columnas debe hacerse restando 1 al valor de `g`. Recordemos que en C los *arrays* siempre comienzan desde la posición 0.

Veamos ahora el desarrollo de las funciones `listarXAnio` y `listarXGrado` dentro de las cuales se emiten los listados solicitados.

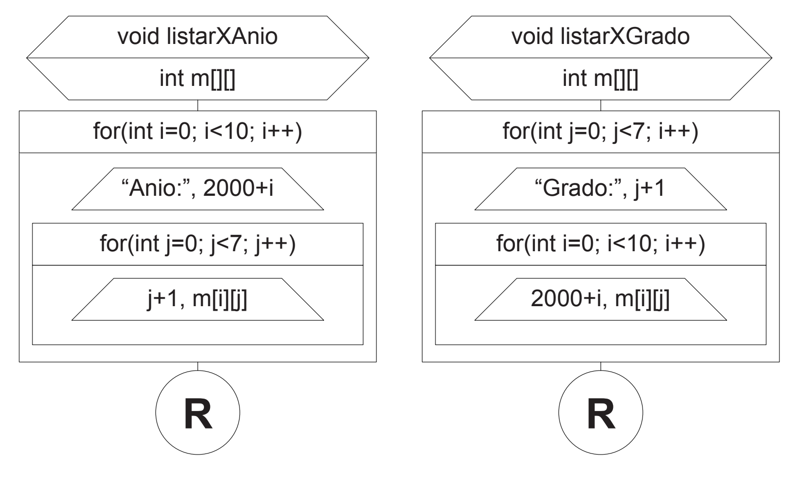


Fig. 7.28 Recorrido por fila/columna y por columna/fila.

7.5.2 Arrays tridimensionales (cubos)

Así como una matriz es un *array*, cuyos elementos también lo son, si definimos una matriz tal que sus celdas contengan *arrays* estaremos ante un *array* tridimensional o un "cubo".

En C definimos un *array* tridimensional de la siguiente manera:

```
int x[3][5][4];
```

La variable `x` representa un cubo compuesto por 3 filas, 5 columnas y 4 planos que, gráficamente, podemos verlo así:

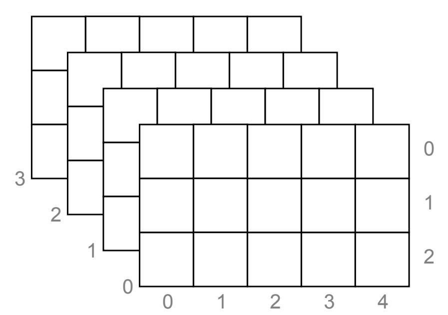


Fig. 7.29 *Array* tridimensional o "cubo".

La siguiente línea de código asigna el valor 10 en la celda ubicada en la fila 2, columna 1, plano 3 del cubo representado en el gráfico:

```
x[2][1][3]=10;
```

7.6 Tipos de datos definidos por el programador

Los lenguajes de programación le permiten al programador crear y definir sus propios tipos de datos. En C los nuevos tipos se definen con la palabra reservada `typedef`.

Veamos un ejemplo:

```
#include <stdio.h>

// defino un nuevo tipo de datos
typedef int Entero;

int main()
{
   // declaro una variable de tipo Entero
   Entero e = 5;

   printf("%d\n",e);

   return 0;
}
```

La posibilidad de definir nuestros propios tipos nos permite crear un nivel de abstracción sobre el cual podemos ocultar gran parte de la complejidad que emerge de las técnicas que utilizamos para resolver algoritmos.

7.6.1 Introducción al encapsulamiento a través de TADs

En los capítulos anteriores, utilizamos variables de tipo `long` para almacenar valores numéricos que representan fechas. Por ejemplo, el entero 20100524 representa la fecha 24 de mayo de 2010. El número tiene 8 dígitos donde los primeros 4 representan el año, los siguientes 2 representan el mes y los dos últimos representan el día.

Si definimos el tipo de datos `Fecha` como un `long` y además, programamos un conjunto de funciones que nos permitan operar y manipular datos de este tipo entonces simplemente podremos pensar en las fechas como variables de tipo `Fecha` y abstraernos del problema de manipular los dígitos del número entero que la almacena.

Para facilitar la comprensión de la idea, analizaremos el siguiente programa:

testFecha.c

```
#include "fecha.h"
#include <stdio.h>

int main()
{
   // "creo" las fechas 2/10/1970 y 3/8/2010
   Fecha f1 = crearFecha(2,10,1970);
   Fecha f2 = crearFecha(3,8,2010);

   // obtengo sus representaciones con formato "dd/mm/aaaa"
   char* sF1 = toString(f1);
   char* sF2 = toString(f2);

   // las comparo para ver cual es posterior
   if( compararFechas(f1,f2)<0 )
   {
      printf("%s es posterior a %s\n",sF2,sF1);
   }
   else
   {
      printf("%s es posterior a %s\n",sF1,sF2);
   }

   return 0;
}
```

En este programa operamos con fechas de la misma forma que lo hicimos en los primeros capítulos. Sin embargo, la claridad y la legibilidad del código es muy superior gracias a que utilizamos el tipo de datos `Fecha` y sus funciones asociadas: `crearFecha`, `toString` y `compararFechas`.

Notemos que en la función `main` “no conocemos” concretamente cómo se implementan las fechas. No sabemos si son números enteros de 8 dígitos, cadenas de caracteres, etc., y tampoco nos debería interesar saberlo ya que toda esa complejidad está encapsulada dentro de las funciones asociadas al nuevo tipo de datos.

El tipo de datos `Fecha` y los prototipos de sus funciones asociadas están definidos en el archivo `fechas.h` cuyo código vemos a continuación:

```
// defino el tipo Fecha
typedef long Fecha;

// "crea" una Fecha
Fecha crearFecha(int,int,int);

// dada una Fecha retorna el dia
int obtenerDia(Fecha);

// dada una Fecha retorna el mes
int obtenerMes(Fecha);

// dada una Fecha retorna el Anio
int obtenerAnio(Fecha);

// dadas dos Fechas f1 y f2 retorna un valor que sera
// mayor, menor o igual a cero segun f2 lo sea respecto de f1
int compararFechas(Fecha,Fecha);

// dada una Fecha retorna una cadena con formato "dd/mm/aaaa"
char* toString(Fecha f);
```

Por último, la implementación de las funciones es la siguiente:

fecha.c

```
#include "fecha.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

Fecha crearFecha(int d,int m ,int a)
{
   return a*10000+m*100+d;
}

int obtenerDia(Fecha f)
{
   return f%100;
}

int obtenerMes(Fecha f)
{
   return (f%10000)/100;
}

int obtenerAnio(Fecha f)
{
   return f/10000;
}

int compararFechas(Fecha f1,Fecha f2)
{
   return f1-f2;
}
```

```
char* toString(Fecha f)
{
   int dia=obtenerDia(f);
   int mes=obtenerMes(f);
   int anio=obtenerAnio(f);

   char* s=(char*)malloc(11);
   sprintf(s,"%02d/%02d/%04d",dia,mes,anio);

   return s;
}
```

■

Al conjunto compuesto por un tipo de datos más las funciones provistas para operar y manipular datos de ese tipo lo llamamos "Tipo Abstracto de Dato" o simplemente TAD. Este tema lo estudiaremos en detalle más adelante.

7.6.2 Estructuras o registros

Una estructura representa un conjunto de variables, probablemente de diferentes tipos, cuyos valores están relacionados entre sí, razón por la cual resulta conveniente tratarlos como unidad. Por ejemplo: un par de coordenadas que marcan un punto dentro de un plano, una terna de enteros que indican el día, mes y año de una fecha o la matrícula, el nombre y la fecha de ingreso de un empleado de una compañía.

En C las estructuras se definen con la palabra reservada `struct`, en cambio, en Pascal se definen con la palabra reservada `record`. Por eso hablamos de "estructuras" o "registros" y cualquiera de los dos términos es comúnmente aceptado.

Las siguientes líneas de código muestran dos formas de definir la estructura `Empleado` descripta más arriba.

Sin usar `typedef`

```
struct Empleado
{
   int matricula;
   char nombre[20];
   Fecha fechaIngreso;
};
```

Usando `typedef`

```
typedef struct Empleado
{
   int matricula;
   char nombre[20];
   Fecha fechaIngreso;
}Empleado;
```

Las dos formas de definir la estructura son correctas. La primera (sin `typedef`) define el tipo de datos `struct Empleado`. En cambio, la segunda define dos tipos de datos: `struct Empleado` y `Empleado`. En general, utilizaremos la segunda opción.

Luego de la definición de la estructura y su correspondiente `typedef` podemos declarar y utilizar variables de este nuevo tipo de datos de la siguiente manera:

```
// declaro una variable de tipo Empleado
Empleado e;

e.matricula=31234;                    // asigno la matricula
strcpy(e.nombre, "Juan");             // asigno el nombre
e.fechaIngreso=crearFecha(21,6,1992); // asigno la fecha
```

Las variables que componen una estructura se llaman "campos". Así, la estructura `Empleado` está compuesta los campos `matricula`, `nombre` y `fechaIngreso`.

7.6.3 Representación gráfica de una estructura

Gráficamente, podemos representar una estructura de la siguiente manera:

Empleado

matricula	nombre	fechaIngreso
int	char[20]	Fecha

Fig. 7.30 Representación gráfica de la estructura Empleado.

El gráfico representa a la estructura Empleado definida más arriba.

7.6.4 Estructuras anidadas

Cuando el tipo de datos de algún campo de la estructura es en sí otra estructura decimos que son estructuras anidadas.

En el siguiente ejemplo, definimos la estructura Dirección cuyos campos permiten almacenar los datos que componen una dirección postal. Luego definimos la estructura Persona cuyo campo dirección es de tipo Direccion.

```
typedef struct Direccion
{
   char calle[50];
   int numero;
   int piso;
   char depto;
}Direccion;

typedef struct Persona
{
   char nombre[30];
   long dni;
   Direccion direccion;
}Persona;
```

Persona y Dirección son estructuras anidadas. Luego, si declaramos una variable de tipo Persona y queremos asignar valores a sus campos podemos hacerlo de la siguiente manera:

```
// declaro una variable de tipo Direccion
Direccion d;
strcpy(d.calle,"Los Patos");
d.numero=222;
d.depto='A';
d.piso=12;

// declaro una variable de tipo Persona
Persona p;
strcpy(p.nombre,"Pablo"); // asigno el nombre
p.dni=23354212;           // asigno el DNI
p.direccion=d;            // asigno la direccion
```

La representación gráfica de la estructura Persona es la siguiente:

Persona

nombre	dni	direccion			
		calle	numero	depto	piso
char[30]	long	char[50]	int	char	int

Fig. 7.31 Representación gráfica de estructuras anidadas.

7.6.5 Estructuras con campos de tipo array

La siguiente estructura incluye un campo de tipo `int[]`.

```
typedef struct Alumno
{
   int matricula;      // matricula del alumno
   char nombre[30];    // nombre
   int notas[3];       // notas obtenidas en los examenes
}Alumno;
```

Podríamos representarla de la siguiente manera:

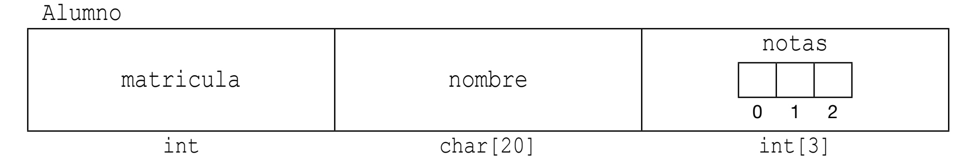


Fig. 7.31a Representación de una estructura con un campo de tipo array.

Luego, con las siguientes líneas de código asignamos el valor 10 en la segunda nota de un alumno.

```
Alumno a;
a.notas[1]=10;
```

7.6.6 Punteros a estructuras

Sean las siguientes líneas de código:

```
Empleado e;
Empleado* p = &e;
```

Para asignar un valor en el campo `matricula` de la estructura que está siendo direccionada por `p` tendremos que hacer:

```
// asigno 10 a la matricula del empleado
(*p).matricula = 10;
```

Como vemos, fue necesario utilizar paréntesis para indicar que el operador de indirección `*` aplica sobre el puntero `p`. Esta situación es habitual cuando, en una función, necesitamos modificar los valores de los campos de una estructura que se recibe por referencia.

Por ejemplo, en la siguiente función recibimos un puntero a una estructura de tipo `Empleado` y asignamos valores en todos sus campos.

```
void cargarEmpleado(Empleado* e)
{
   (*e).matricula=10;
   strcpy((*e).nombre,"Pablo");
   (*e).fechaIngreso=crearFecha(2,10,2009);
}
```

7.6.6.1 El operador "flecha" ->

El operador `->` (léase “flecha”) permite hacer referencia directa a los campos de una estructura que es accedida a través de un puntero. Utilizando este operador, la función anterior podría reescribirse de la siguiente manera:

```
void cargarEmpleado(Empleado* e)
{
   e->matricula=10;
   strcpy(e->nombre,"Pablo");
   e->fechaIngreso=crearFecha(2,10,2009);
}
```

7.6.7 Arrays de estructuras

En ocasiones resulta muy práctico trabajar con *arrays* de estructuras. En la siguiente línea, declaramos un *array* con capacidad para contener 100 empleados.

```
Empleado aEmp[100];
```

Podemos asignarle valores de la siguiente manera:

```
int len=0;
agregarEmpleado(aEmp,&len,23213,"Juan",crearFecha(2,12,2010));
agregarEmpleado(aEmp,&len,11243,"Pablo",crearFecha(5,10,2008));
agregarEmpleado(aEmp,&len,10021,"Pedro",crearFecha(22,4,2005));
```

donde el código de la función `agregarEmpleado` es:

```
void agregarEmpleado(Empleado emps[], int* len, int leg, char* nom, Fecha f)
{
   emps[*len].matricula=leg;
   strcpy(emps[*len].nombre,nom);
   emps[*len].fechaIngreso=f;
   *len++;
}
```

Y gráficamente podemos visualizarlo así:

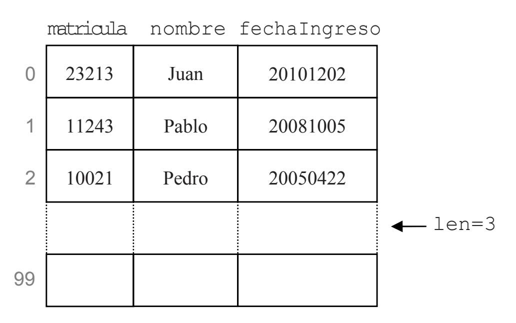


Fig. 7.32 *Array* de tipo `Empleado`.

7.6.8 Estructuras con campos de tipo "array de estructuras"

Modificaremos la estructura `Alumno` para que incluya un campo de tipo `Nota[]`.

```
typedef struct Nota
{
   int puntaje;
   long fecha;
}Nota;


typedef struct Alumno
{
   int matricula;   // matricula del alumno
   char nombre[30]; // nombre
   Nota notas[3];   // notas obtenidas en los examenes
}Alumno;
```

Podemos representarla de la siguiente manera:

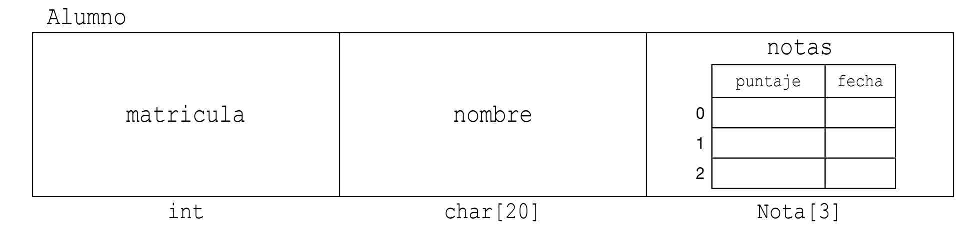


Fig. 7.33 Estructura con un campo de tipo *array* de estructura.

En las siguientes líneas, asignamos el valor 10 en la segunda nota de un alumno.

```
Alumno a;
a.notas[1].puntaje=10;
```

7.7 Resumen

En este capítulo, estudiamos operaciones sobre *arrays*: agregar, insertar, insertar en orden, eliminar y buscar elementos dentro de un *array*. Estudiamos también los algoritmos de la búsqueda binaria y el ordenamiento burbuja. Todas estas operaciones las utilizaremos, más adelante, para resolver problemas en el capítulo integrador.

Vimos que podemos definir nuestros propios tipos de datos y cómo esta posibilidad puede ser usada para encapsular (ocultar) cierta lógica de la implementación del algoritmo.

En el próximo capítulo, analizaremos estructuras de datos externas: archivos en los que podremos almacenar información para que persista más allá del programa que la grabó.

7.8 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

7.8.1 Mapa conceptual

7.8.2 Autoevaluaciones

7.8.3 Videotutoriales

7.8.3.1 Algoritmo de la burbuja

7.8.3.2 Algoritmo de la búsqueda binaria

7.8.4 Presentaciones*

8

Operaciones sobre archivos

Contenido

Objetivos del capítulo

- Comprender el concepto de “archivo secuencial” y operaciones básicas asociadas.
- Entender la existencia y uso del “indicador de posición”.
- Diferenciar correctamente los conceptos: “archivo de texto”, “archivo binario” y “archivo de registros”.
- Conocer las limitaciones de los archivos secuenciales.
- Desarrollar operaciones lógicas sobre archivos de registros.
- Establecer una relación unívoca entre el “byte actual” y el “registro actual”.
- Diseñar técnicas de “baja lógica” de registros.
- Usar las funciones que provee la biblioteca estándar de C: `fread`, `fwrite`, `fopen`, `fclose`, `feof`, `fseek`, `ftell` y otras.
- Valorizar el uso *buffers* para reducir el *overhead* de las operaciones de I/O.

Competencias específicas

- Aplicar los conocimientos adquiridos en la construcción de un programa que almacena y manipula datos por medio de archivos.

8.1 Introducción

Llamamos "estructura de datos externa" a toda colección de datos almacenada en cualquier dispositivo que no sea la memoria principal o memoria RAM de la computadora. A este tipo de colección de datos lo llamamos "archivo".

Físicamente, un archivo no es más que una sucesión de *bytes* almacenados en algún medio de almacenamiento como ser un disco duro, un *pendrive*, un CD o un DVD, pero desde el punto de vista lógico esta colección de *bytes* representa información relacionada con las entidades que conforman el contexto de nuestra aplicación. Por ejemplo: "clientes", "productos", "personas", "empleados", "facturas", etcétera.

El objetivo de este capítulo es introducir al lector en las operaciones físicas de manejo de archivos de forma tal que pueda comprender los conceptos de "archivo secuencial", "archivo de acceso directo", "archivo de registros" y las diferencias que existen entre un "archivo de texto" y un "archivo binario".

Aquí estudiaremos las operaciones básicas de apertura/cierre y lectura/escritura. Los aspectos lógicos y de aplicación relacionados con archivos los estudiaremos en los capítulos siguientes.

8.1.1 Memoria principal o memoria RAM de la computadora

La "memoria principal" o "memoria RAM" de la computadora es volátil, lo que significa que los datos que allí se almacenan se perderán cuando finalice la ejecución del programa. Si necesitamos que cierta información perdure más allá del programa que la está gestionando tenemos que almacenarla en algún medio de almacenamiento.

8.1.2 Medios de almacenamiento (memoria secundaria)

Los medios de almacenamiento son aquellos soportes físicos en los cuales podemos almacenar información. Por ejemplo, un disco rígido, un DVD o CD, una cinta, son soportes externos, físicos y no volátiles que permiten almacenar información en forma persistente y, de esta manera, trascendente a la ejecución del programa.

Al tratarse de dispositivos externos y mecánicos el tiempo que requieren para acceder a la información es mucho mayor que el tiempo que demanda la memoria RAM. Esto nos obliga a ser prudentes en su uso y a minimizar las operaciones de entrada/salida tanto como sea posible ya que, en parte, de esto dependerá el rendimiento de la aplicación.

8.2 Archivos

Como mencionamos más arriba, un archivo representa una sucesión de *bytes* almacenados en algún medio de almacenamiento. Esto hace que la información que contiene sea persistente y trascienda a la ejecución del programa.

Desde un programa podemos crear archivos para almacenar información o bien abrir archivos que fueron creados por otros programas para leer y manipular su contenido.

Antes de usar un archivo tenemos que abrirlo y luego de usarlo tenemos que cerrarlo. Mientras el archivo se encuentre abierto podemos acceder a sus datos, modificarlos y agregar más datos al final. La biblioteca estándar de C provee funciones específicas que nos permiten llevar a cabo todas estas operaciones.

Para abrir un archivo utilizamos la función `fopen` y para cerrarlo usamos la función `fclose`. Para escribir datos en el archivo disponemos de la función `fwrite` y para leer datos desde un archivo contamos con la función `fread`. Los archivos se representan con variables de tipo `FILE*`.

8.2.1 Abrir un archivo

Para abrir un archivo utilizamos la función `fopen`. Esta función recibe el nombre físico del archivo y una cadena que indica la modalidad de apertura.

Modalidad de apertura	Descripción
"w+b"	Crea un archivo sobre el que podemos grabar registros y, leer y modificar registros ya grabados. Si dentro de la carpeta existe un archivo con igual nombre entonces, al abrirlo, borrará todo su contenido dejándolo con 0 *bytes*.
"r+b"	Idem anterior solo que si en la carpeta ya existe un archivo con igual nombre entonces no borrará su contenido.

Existen otras modalidades de apertura, pero por el momento no las consideraremos.

8.2.2 Escribir datos en un archivo

Analizaremos el código de un programa que crea un archivo (inicialmente inexistente) para grabar tres caracteres. Utilizamos las funciones `fopen`, `fwrite` y `fclose`.

```
#include <stdio.h>

int main()
{
   // abro el archivo
   FILE* arch=fopen("DEMO.dat","w+b");

   char c;

   // escribo una 'A'
   c='A';
   fwrite(&c,sizeof(char),1,arch);

   // escribo una 'B'
   c='B';
   fwrite(&c,sizeof(char),1,arch);

   // escribo una 'C'
   c='C';
   fwrite(&c,sizeof(char),1,arch);

   // cierro el archivo
   fclose(arch);

   return 0;
}
```

Luego de ejecutar este programa se creará, dentro de la misma carpeta, el archivo `DEMO.dat` cuyo contenido serán los caracteres: ‘A’, ‘B’ y ‘C’. Si abrimos el archivo con algún editor de texto lo podremos verificar.

Analicemos ahora el código del programa, línea por línea.

Comenzamos declarando la variable `arch` de tipo `FILE*` a la que le asignamos el valor de retorno de la función `fopen` de la siguiente manera:

```
// abro el archivo
FILE* arch=fopen("DEMO.dat","w+b");
```

La función `fopen` recibe dos argumentos: el primero es el nombre del archivo físico con el que vamos a trabajar. El segundo es una cadena que indica la modalidad en la cual lo queremos abrir.

Dado que vamos a grabar caracteres necesitamos una variable de tipo `char`.

```
char c;
```

Luego utilizamos la función `fwrite` para grabar los caracteres en el archivo que está siendo representado por la variable `arch`.

```
// escribo una 'A'
c='A';
fwrite(&c,sizeof(char),1,arch);
```

La función `fwrite` recibe un puntero a la variable que contiene el dato que vamos a grabar (`&c`), luego recibe la longitud de ese tipo de datos (valor que obtenemos invocando a `sizeof`), la cantidad de datos de ese tipo que queremos grabar y, por último, el archivo. Cada vez que invocamos a `fwrite` estamos grabando un carácter al final del archivo.

Luego de grabar los caracteres 'B' y 'C' cerramos el archivo con la función `fclose`.

```
fclose(arch);
```



Leer y escribir un archivo

8.2.3 Leer datos desde un archivo

Ahora analizaremos el código de un programa que lee el archivo recientemente creado y muestra por pantalla cada uno de sus caracteres.

En este programa, utilizaremos las funciones `fopen` y `fclose` (ya conocidas) y las funciones `fread` y `feof`. Esta última nos permitirá determinar si llegamos al final del archivo o no (*eof* son las iniciales de "*end of file*" o, en español: "fin de archivo").



```
#include <stdio.h>

int main()
{
   FILE* arch;
   char c;

   // abro el archivo
   arch = fopen("DEMO.dat","r+b");

   // leo el primer caracter
   fread(&c,sizeof(char),1,arch);

   // y mientras no llegue al final del archivo...
   while( !feof(arch) )
   {
      // muestro el caracter leido
      printf("%c\n",c);

      // leo el siguiente caracter
      fread(&c,sizeof(char),1,arch);
   }

   // cierro el archivo
   fclose(arch);

   return 0;
}
```

En este caso, abrimos el archivo para leer su contenido, no para escribirlo. Por lo tanto, la cadena de "modalidad de apertura" que le pasamos a `fopen` es: `"r+b"`.

```
arch = fopen("DEMO.dat","r+b");
```

La función `fread` recibe los mismos argumentos que `fwrite`. Luego, lee un carácter desde el archivo y lo asigna en la variable `c`.

Cada vez que invocamos a `fread` accederemos al siguiente carácter. Así hasta que lleguemos al final del archivo. En este caso, la función `feof` retornará *true* y el ciclo `while` dejará de iterar.

La salida del programa será la siguiente:

```
A
B
C
```

8.2.4 El identificador de posición (puntero)

El tipo `FILE` almacena un identificador de posición o "puntero" que hace referencia al próximo *byte* que debe ser tratado dentro del archivo.

Por ejemplo, `DEMO.dat` contiene tres caracteres. El primero está ubicado en el *byte* número 0, el segundo está ubicado en el *byte* número 1 y el último carácter se encuentra ubicado en el *byte* número 2.

Al abrir el archivo con la función `fopen`, el identificador de posición apuntará al próximo el *byte* que debe ser tratado. Esto es: el *byte* número cero (el primer *byte*).

fopen(arch, "r+b")

A	B	C	*eof*
0	1	2	

Luego, la función `fread` nos permitirá acceder al carácter apuntado por el identificador de posición (que en este caso es 'A') e incrementará el valor de dicho identificador haciéndolo apuntar al *byte* número 1 ya que este es el próximo *byte* que se deberá tratar.

fread(...)

A	B	C	*eof*
0	1	2	

Si volvemos a invocar a `fread` accederemos al carácter 'B' y el indicador de posición apuntará al *byte* número 2.

fread(...)

A	***B***	C	*eof*
0	1	2	

Una nueva lectura nos permitirá acceder al carácter 'C' y colocará al identificador de posición al final del archivo (*eof*).

fread(...)

A	B	***C***	*eof*
0	1	2	

En esta situación, la función `feof` retornará *true*, lo que nos permitirá evitar realizar una nueva lectura que, obviamente, arrojará un error.

8.2.5 Representación gráfica

Escribir datos en un archivo implica una operación de salida. Análogamente, leer desde un archivo implica una operación de entrada. Por esto, para representar gráficamente los dos programas anteriores utilizaremos los símbolos de entrada y salida.

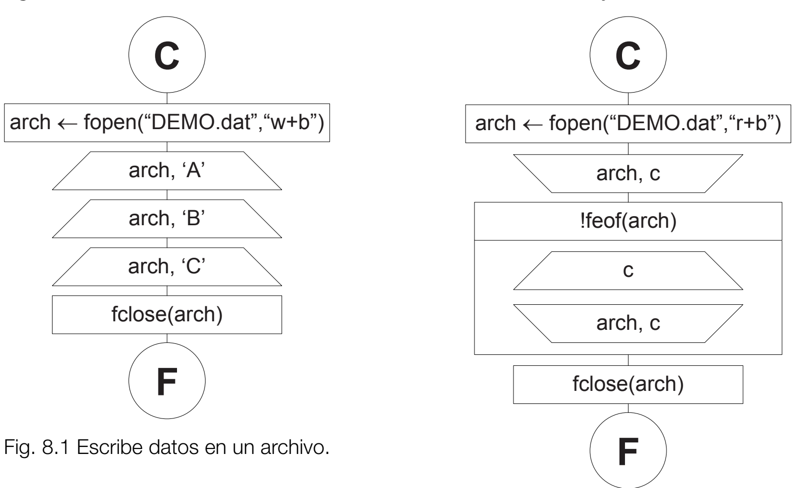
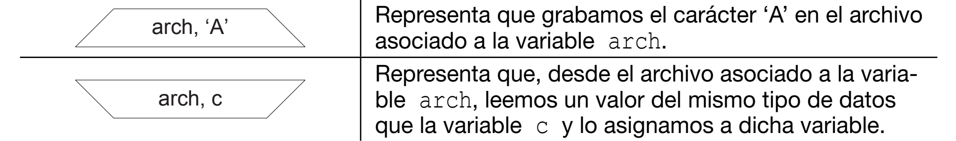


Fig. 8.1 Escribe datos en un archivo.

Fig. 8.2 Lee datos desde un archivo.

Notemos que en los símbolos de entrada y salida solo indicamos los datos que son relevantes para la comprensión del diagrama. Esto es:



arch, 'A'	Representa que grabamos el carácter 'A' en el archivo asociado a la variable `arch`.
arch, c	Representa que, desde el archivo asociado a la variable `arch`, leemos un valor del mismo tipo de datos que la variable `c` y lo asignamos a dicha variable.

8.2.6 Valor actual del identificador de posición (función `ftell`)

La biblioteca estándar de C provee la función `ftell` que retorna el número de *byte* que está siendo apuntado por el identificador de posición del archivo. La utilizaremos para hacer que el programa anterior muestre, además, el número de *byte* (o posición) en el que se encuentra grabado cada carácter.

```
#include <stdio.h>

int main()
{
   FILE *arch;
   char c;

   // abro el archivo
   arch = fopen("DEMO.dat", "r+b");

   // obtengo la posicion actual
   long pos=ftell(arch);
```

```
    // leo el primer caracter
    fread(&c, sizeof(char), 1, arch);

    // y mientras no llegue al final del archivo...
    while( !feof(arch) )
    {
       // muestro el caracter leido
       printf("byte numero: %ld, %c\n",pos, c);

       // obtengo la posicion actual
       pos=ftell(arch);

       // leo el siguiente caracter
       fread(&c, sizeof(char), 1, arch);
    }

    // cierro el archivo
    fclose(arch);

    return 0;
}
```
■

La salida será:

```
byte numero: 0, A
byte numero: 1, B
byte numero: 2, C
```

8.2.7 Manipular el valor del identificador de posición (función `fseek`)

La función de biblioteca `fseek` permite modificar, arbitrariamente, el valor del identificador de posición del archivo para hacerlo apuntar a cualquiera de sus *bytes*. La función recibe tres argumentos:

- El archivo (de tipo `FILE*`).
- El número de *byte* al cual queremos “mover” el puntero (lo llamaremos `pos`).
- Un entero cuyos valores posibles se definen con las siguientes constantes:

`SEEK_SET`	Indica que el número de *byte* (`pos`) es absoluto, contando desde el inicio del archivo.
`SEEK_CUR`	Indica que el número de *byte* (`pos`) es relativo al valor actual del identificador de posición.
`SEEK_END`	Indica que el número de *byte* (`pos`) es absoluto, contando desde el final del archivo.

Así, con la siguiente línea de código movemos el identificador de posición del archivo `arch` para hacerlo apuntar al primer *byte* (el *byte* número cero).

```
fseek(arch,0,SEEK_SET);
```

Y en la siguiente línea hacemos que el identificador de posición apunte al final del archivo.

```
fseek(arch,0,SEEK_END);
```

Ahora, si el identificador de posición está apuntando al *byte* número *n* entonces con la siguiente línea de código lo haremos apuntar al *byte* número *n*+1.

```
fseek(arch,1,SEEK_CUR);
```

y con la siguiente línea de código lo haremos apuntar al *byte* número *n*-1.

```
fseek(arch,-1,SEEK_CUR);
```

8.2.8 Calcular el tamaño de un archivo

Con las funciones `ftell` y `fseek` podemos calcular el tamaño (expresado en *bytes*) de un archivo. Desarrollaremos entonces la función `fileSize` que recibe un `FILE*` y retorna su longitud.

Básicamente, la idea es mover el identificador de posición al final del archivo para averiguar cuál es el número de su último *byte* y luego volver a posicionarlo en el *byte* al que estaba apuntando antes de haberlo movido.

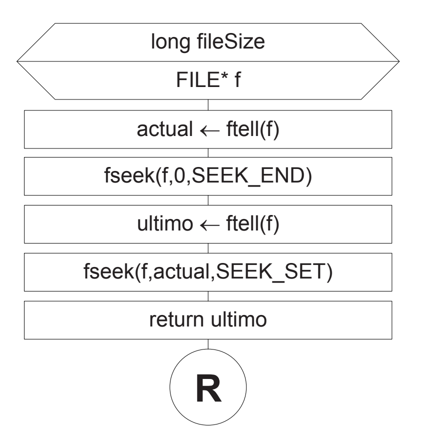


Fig. 8.3 Obtener la longitud de un archivo.

Veamos la codificación de `fileSize` y luego un programa que la utiliza para mostrar el tamaño del archivo `DEMO.dat` utilizado más arriba.

```
#include <stdio.h>

// retorna la longitud (en bytes) de un archivo
long fileSize(FILE* f)
{
   long actual=ftell(f);
   fseek(f,0,SEEK_END);
   long ultimo=ftell(f);
   fseek(f,actual,SEEK_SET);
   return ultimo;
}

int main()
{
   FILE *arch;
   char c;

   // abro el archivo
   arch = fopen("DEMO.dat", "r+a");

   long size=fileSize(arch);
   printf("El archivo tiene %ld bytes\n",size);

   return 0;
}
```

La salida será:

```
El archivo tiene 3 bytes
```

8.2.9 Archivos de texto vs. archivos binarios

Como mencionamos anteriormente, los archivos son sucesiones de *bytes*. Si cada uno de estos *bytes* constituye en sí mismo una unidad de información y su valor numérico coincide con el código ASCII de algún carácter diremos que el archivo es un "archivo de texto". De lo contrario, diremos que se trata de un "archivo binario".

Podemos decir entonces que "archivo de texto" es un caso particular de "archivo binario".

En el programa anterior, grabamos caracteres de la siguiente manera:

```
char c='A';
fwrite(&c,sizeof(char),1,arch);
```

Cada carácter que grabamos agrega un *byte* al final del archivo cuya representación numérica es su código ASCII.

Luego de grabar los caracteres: 'A', 'B', y 'C' el archivo tendrá el siguiente contenido:

Binario	01000001	01000010	01000011
Decimal	65	66	67
Carácter	'A'	'B'	'C'

Como podemos ver, el valor numérico de cada uno de los *bytes* del archivo coincide con el código ASCII del carácter que representa. Por lo tanto, podemos decir que el archivo `DEMO.dat` es un archivo de texto y, en consecuencia, podremos ver y editar su contenido con cualquier editor.

Por lo general, se estila que los nombres de los archivos de texto finalicen con la extensión ".txt", pero esto no es excluyente. Lo que determina si un archivo es o no un archivo de texto es su contenido, no su nombre y/o extensión.

¿Qué diferencia tendrá el archivo que generamos con el programa anterior si, en lugar de grabar caracteres (tipo `char`), grabamos enteros (tipo `int`)? Veamos:

```
int i='A';
fwrite(&i,sizeof(int),1,arch);
```

Aquí asignamos a la variable `i` el valor ASCII del carácter 'A'. Es decir, asignamos en `i` el valor numérico 65 y lo grabamos. Recordemos que por cuestiones didácticas aceptamos que el tipo de datos `int` se representa con 2 *bytes*. Luego, si grabamos los valores 65, 66 y 67 como `int` el archivo resultante será el siguiente:

Binario	00000000	01000001	00000000	01000010	00000000	01000011
Decimal	65		66		67	
Carácter	'[ ]'	'A'	'[ ]'	'B'	'[ ]'	'C'

Si bien ambos archivos contienen la misma información, en este caso cada carácter está codificado en 2 *bytes*. El archivo, ahora, es un archivo binario y si lo abrimos con un editor de texto veremos símbolos extraños (representando los *bytes* 00000000) intercalados con los caracteres 'A', 'B' y 'C'.

Este tema lo profundizaremos más adelante.

8.3 Archivos de registros

Cuando en la sucesión de *bytes* que componen el archivo podemos reconocer la existencia de un determinado patrón decimos que el archivo contiene registros o bien que el archivo es "un archivo de registros".

En el punto anterior, donde analizamos el archivo que se hubiera generado al grabar los valores 65, 66 y 67 como datos enteros, llegamos a esta conclusión:

Binario	00000000	01000001	00000000	01000010	00000000	01000011
Decimal	65		66		67	
Carácter	‘ []’	‘A’	‘ []’	‘B’	‘ []’	‘C’

Esta sucesión de *bytes* responde a un patrón determinado: cada 2 *bytes* se codifica un `int`. Podemos decir entonces que este es un archivo de registros donde cada registro es un `int`.

8.3.1 Archivos de estructuras

En general, cuando hablamos de “archivos de registros” nos referimos a archivos cuyos registros son estructuras. A continuación, definimos la estructura `Alumno` y luego dos programas con los que grabaremos alumnos en un archivo para, más tarde, leerlos y mostrarlos por consola.

alumnos.h

```
typedef struct Alumno
{
    int matricula;
    char nombre[20];
    int nota;
}Alumno;
```

8.3.1.1 Grabar estructuras (registros) en un archivo

El siguiente programa graba tres “alumnos” en un archivo.

grabaAlumnos.c

```
#include <stdio.h>
#include <string.h>
#include "alumnos.h"

Alumno crearAlumno(int,char[],int);

int main()
{
   // abro el archivo
   FILE* arch=fopen("ALUMNOS.dat","w+b");

   Alumno a;

   // grabo un alumno
   a=crearAlumno(10,"Pablo",7);
   fwrite(&a,sizeof(Alumno),1,arch);


   // grabo un alumno
   a=crearAlumno(20,"Juan",5);
   fwrite(&a,sizeof(Alumno),1,arch);
```

```
   // grabo un alumno
   a=crearAlumno(30,"Pedro",8);
   fwrite(&a,sizeof(Alumno),1,arch);

   // cierro el archivo
   fclose(arch);

   return 0;
}

Alumno crearAlumno(int matr,char nom[], int nota)
{
   Alumno x;
   x.matricula=matr;
   strcpy(x.nombre,nom);
   x.nota=nota;
   return x;
};
```

Luego de ejecutar este programa se creará, dentro de la misma carpeta, el archivo `ALUMNOS.dat` cuyo contenido serán los datos de los tres alumnos que grabamos.

Como vemos, el programa es prácticamente idéntico al programa con el que grabamos el archivo de caracteres. Por cuestiones de comodidad y claridad de código, desarrollamos la función `crearAlumno` que recibe los datos de un alumno y retorna una estructura con todos estos datos asignados en sus campos.



Leer y escribir un archivo de registros

8.3.1.2 Leer estructuras (registros) desde un archivo

El siguiente programa lee cada uno de los registros contenidos en `ALUMNOS.dat` y los muestra por pantalla.

leeAlumnos.c

```
#include <stdio.h>
#include <string.h>
#include "alumnos.h"

int main()
{
   FILE* arch;
   Alumno a;

   // abrimos el archivo para lectura
   arch = fopen("ALUMNOS.dat","r+b");

   // la primer leida la hacemos afuera del while
   fread(&a,sizeof(Alumno),1,arch);
   // iteramos mientras no sea eof
   while( !feof(arch) )
   {
      printf("%d, %s, %d\n",a.matricula,a.nombre,a.nota);
```

```
      // leemos el siguiente registro del archivo
      fread(&a,sizeof(Alumno),1,arch);
   }

   // cerramos el archivo
   fclose(arch);

   return 0;
}
```

La salida de este programa será:

```
10, Pablo, 7
20, Juan, 5
30, Pedro, 8
```

8.3.1.3 Legibilidad del código fuente

Las funciones `fread` y `fwrite` pueden resultar incómodas de invocar e, incluso, incómodas de ver dentro del código fuente de un programa. Por este motivo, vamos a desarrollar dos funciones que simplificarán su uso.

```
void leerAlumno(FILE* arch, Alumno* reg)
{
   fread(reg,sizeof(Alumno),1,arch);
}

void grabarAlumno(FILE* arch, Alumno* reg)
{
    fwrite(reg,sizeof(Alumno),1,arch);
}
```

Luego, en lugar de invocar a `fread` podemos invocar a `leerAlumno` así:

```
leerAlumno(arch, &a);
```

y en lugar de invocar a `fwrite` podemos invocar a `grabarAlumno` así:

```
grabarAlumno(arch, &a);
```

El resultado será el mismo, pero la legibilidad del programa habrá mejorado.

```
// :
int main()
{
   // :

   // la primer lectura la hacemos afuera del while
   leerAlumno(arch,&a);
```

```
   // iteramos mientras no sea eof
   while( !feof(arch) )
   {
      printf("%d, %s, %d\n",a.matricula,a.nombre,a.nota);

      // leemos el siguiente registro del archivo
      leerAlumno(arch,&a);
   }

   // :

   return 0;
}
```

Análogamente, el lector puede desarrollar las funciones: `leerChar`, `grabarChar`, `leerInt` y `grabarInt`.

La representación gráfica de estos programas podría ser la siguiente:

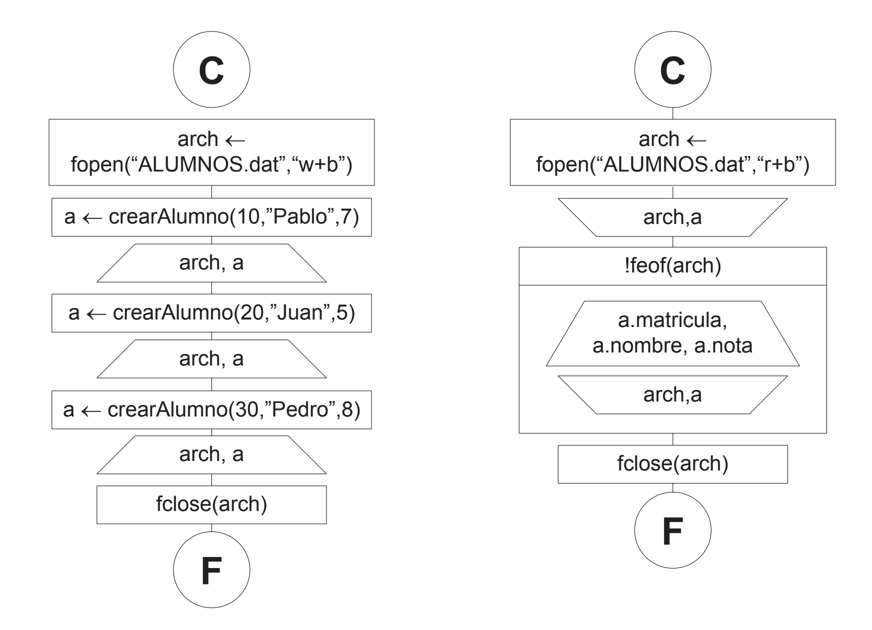


Fig. 8.4 Graba alumnos en un archivo.

Fig. 8.5 Lee alumnos desde un archivo y los muestra por pantalla.

8.3.2 Acceso directo a registros

Como comentamos más arriba, el tipo `FILE` mantiene un identificador de posición con la referencia al próximo *byte* que debe ser procesado. La función `fseek` nos permite manipular arbitrariamente su valor y, de esta forma, acceder directamente a cualquier *byte* del archivo para leerlo o escribirlo (modificarlo).

Recordemos la estructura del archivo `ALUMNOS.dat`:

```
typedef struct Alumno
{
    int matricula;    // 2 bytes
    char nombre[20];  // 20 bytes
    int nota;         // 2 bytes
}Alumno;
```

Cada registro que contiene el archivo ocupa 24 *bytes*: 2 *bytes* para la matrícula, 20 *bytes* para el nombre y 2 *bytes* la nota. Esto significa que el primer registro está ubicado entre los *bytes* 0 y 23. El segundo comienza en el *byte* 24 y finaliza en el *byte* 47 y, en general, el *i-ésimo* registro comienza en el *byte* *i*`*sizeof(`*T*`)` donde *T* es el tipo de datos del registro.

8.3.2.1 Acceso directo para lectura

En el siguiente programa, le pedimos al usuario que ingrese un número de registro. Luego hacemos un acceso directo para leer el contenido del registro y mostrarlo por pantalla.

```
#include <stdio.h>
#include "alumnos.h"

int main()
{
   FILE* arch=fopen("ALUMNOS.dat","r+b");

   int n;
   printf("Ingrese numero de registro: ");
   scanf("%d",&n);

   // posiciono el puntero del archivo
   fseek(arch,n*sizeof(Alumno),SEEK_SET);

   // con el puntero posicionado, leo el registro
   Alumno reg;
   fread(&reg,sizeof(Alumno),1,arch);

   // muestro los datos…
   printf("Matricula: %d\n",reg.matricula);
   printf("Nombre: %s\n",reg.nombre);
   printf("Nota: %d\n",reg.nota);

   fclose(arch);

   return 0;
}
```

Este programa podemos representarlo gráficamente como se observa en la Fig.8.6.

Notemos que en el diagrama no incluimos demasiado nivel de detalle. La función `fseek` la reemplazamos por una (supuesta) función `seek` que solo recibe como argumentos el nombre del archivo y el número de registro sobre el que nos queremos posicionar.

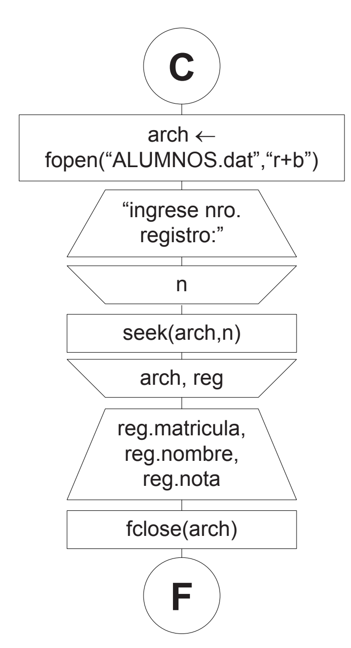


Fig. 8.6 Acceso directo a un registro.

8.3.2.2 Acceso directo para escritura

De la misma manera que ubicamos el identificador de posición para hacerlo apuntar al inicio del registro que queremos leer, ahora lo ubicaremos al inicio del registro sobre el cual queremos escribir. Esto nos permitirá modificar sus datos.

En el siguiente programa, le pedimos al usuario que ingrese el número de registro cuyos datos quiere modificar y luego los nuevos que desea grabar.

```
#include <stdio.h>
#include "alumnos.h"

// prototipo de una funcion que lee x consola los nuevos datos
Alumno ingresoDatosXConsola();

int main()
{
   FILE* arch=fopen("ALUMNOS.dat","r+b");

   int n;
   printf("Ingrese numero de registro: ");
   fflush(stdout);
   scanf("%d",&n);

   // ingreso los nuevos datos por consola
   Alumno reg = ingresoDatosXConsola();

   // posiciono el identificador de posicion
   fseek(arch,n*sizeof(Alumno),SEEK_SET);
```

```
    // grabo el registro pisando los valores anteriores
    fwrite(&reg,sizeof(Alumno),1,arch);

    fclose(arch);
}

Alumno ingresoDatosXConsola()
{
    int matricula,nota;
    char nombre[20];

    printf("Ingrese nuevo matricula: ");
    scanf("%d",&matricula);

    printf("Ingrese nuevo nombre: ");
    scanf("%s",nombre);

    printf("Ingrese nueva nota: ");
    scanf("%d",& nota);

    Alumno a;
    a.matricula=matricula;
    strcpy(a.nombre,nombre);
    a.nota=nota;

    return a;
}
```

8.3.2.3 Agregar un registro al final del archivo

Para agregar un nuevo registro, tenemos que ubicar al identificador de posición al final del archivo. Para esto, utilizaremos la función `fseek` y le pasaremos la constante `SEEK_END` que, justamente, es una referencia al final.

En el siguiente programa, el usuario ingresa los datos de un alumno y, luego, utilizamos estos datos para grabar un registro al final del archivo `ALUMNOS.dat`.

```
#include <stdio.h>
#include "alumnos.h"

// prototipo de la funcion de ingreso de datos
Alumno ingresoDatosXConsola();

int main()
{
    FILE* arch=fopen("ALUMNOS.dat","r+b");

    // ingreso los datos por consola
    Alumno reg = ingresoDatosXConsola();

    // posiciono el puntero al final del archivo
    fseek(arch,0,SEEK_END);
```

```
   // grabo el nuevo registro
   fwrite(&reg,sizeof(Alumno),1,arch);

   fclose(arch);

   return 0;
}

// :
```

Para posicionarnos al final del archivo utilizamos la constante `SEEK_END`.

Registro 1			Registro 2			Registro 3			
0	…	23	24	…	47	48	…	71	*eof*

`SEEK_END` hace referencia al final del archivo que, según este gráfico se encuentra ubicado luego del *byte* 71. Entonce en el *byte* número 0 (contando desde el final del archivo) será la posición donde vamos a grabar el nuevo registro.

8.3.3 Calcular la cantidad de registros que tiene un archivo

Dado que los archivos que estamos estudiando almacenan conjuntos de registros, todos de igual longitud, podemos calcular cuantos registros tiene el archivo dividiendo `fileSize(arch)/sizeof(`*T*`)` donde `arch` es el identificador del archivo y *T* es la estructura o tipo de datos de sus registros.

En las siguientes líneas de código, mostramos la cantidad de registros que tiene el archivo `ALUMNOS.dat`.

```
// abro el archivo
FILE* arch=fopen("ALUMNOS.dat","r+b");

// calculo la cantidad de registros
int cantReg=fileSize(arch)/sizeof(Alumno);

// lo muestro
printf("El archivo tiene %d registros\n",cantReg);
```

8.4 Lectura y escritura en bloques (buffers)

Como explicamos al principio, las operaciones de entrada/salida suelen ser “costosas” si las medimos en tiempo de procesamiento ya que, en general, interactúan con dispositivos mecánicos. Para reducir estos tiempos y minimizar el *overhead* derivado de este tipo de operaciones se utilizan *buffers*.

Para ilustrar el problema, gráficamente, imaginemos la siguiente e hipotética situación:

Contratamos a dos personas para que vayan al bosque a juntar 100 manzanas cada una. La primera persona, a quien llamaremos `A`, va al bosque, recoge una manzana y la trae. Luego vuelve al bosque a buscar la segunda manzana y la también la trae. Luego, repite esta operación hasta completar el pedido de 100 manzanas que le encargamos.

La segunda persona, a quien llamaremos `B`, va al bosque con una canasta con capacidad para cargar 10 manzanas. Luego, cada vez que va al bosque regresa con 10 manzanas en la canasta por lo que el trayecto hacia y desde el bosque lo redujo 10 veces.

¿Cuál de las dos personas que contratamos resultó ser más eficiente?

La respuesta es obvia: `B` fue mucho más eficiente que `A` y, en consecuencia, demoró menos tiempo en concretar el trabajo que se le encargó.

En este contexto, cada viaje al bosque representa el *overhead* asociado a una operación de entrada/salida y la canasta utilizada por `B` representa el *buffer* gracias al cual `B` redujo 10 veces la tarea más pesada.

Las funciones `fread` y `fwrite` (respectivamente) permiten leer y escribir bloques de datos, de cualquier tipo. Es decir: podemos leer 1 *byte* o *n bytes*, podemos leer un registro de tipo `Alumno` o *n* registros de este tipo, etcétera.

Por ejemplo, con las siguientes líneas de código leemos los primeros 1024 *bytes* de un archivo.

```
// :
char buffer[1024];
int n=fread(buffer,sizeof(char),1024,arch);
//:
```

La función `fread` retorna la cantidad de *bytes* que efectivamente han sido leídos y asignados en `buffer`. Es decir: supongamos que el archivo que estamos leyendo tiene 1500 *bytes*. Entonces, en la primera lectura `fread` retornará 1024, pero en la segunda invocación retornará 476 ya que esta es la cantidad de *bytes* remanentes.

Podremos tomar conciencia de la gran importancia que tiene este tema analizando el siguiente programa que, simplemente recorre un archivo y muestra por pantalla el tiempo (expresado en segundos) que demoró en recorrerlo. El nombre del archivo y el tamaño del *buffer* son argumentos que se le deben proporcionar en línea de comandos.

Veamos el código, luego los resultados hablan por sí mismos.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc,char* argv[])
{
   // creo un buffer
   int bufferLen=atoi(argv[2]);
   char *buffer=(char*)malloc(bufferLen);

   // abro el archivo
   FILE* f = fopen(argv[1],"r+b");

   // tomo la hora actual (hora inicial)
   time_t t1=time(NULL);

   // leo bufferLen bytes
   int n=fread(buffer,1,bufferLen,f);

   int i=0;
   while( n==bufferLen )
   {
      i+=bufferLen;
      n=fread(buffer,1,bufferLen,f);
   }
```

```
   // sumo el remanente
   i+=n;

   fclose(f);

   // tomo la hora actual (hora final)
   time_t t2=time(NULL);

   // obtengo la diferenecia entre la hora inicial y la hora final
   double secs=difftime(t2,t1);

   printf("Total bytes leidos: %d\n",i);
   printf("%.0lf segundos, buffer=%d bytes\n",secs,bufferLen);

   return 0;
}
```

Cuatro corridas de este mismo programa utilizando *buffers* de 1, 10, 100 y 1024 *bytes* para recorrer un archivo de aproximadamente 240 MB arrojaron los siguientes resultados:

```
Total bytes leidos: 248237751
28 segundos, buffer=1 bytes

Total bytes leidos: 248237751
3 segundos, buffer=10 bytes

Total bytes leidos: 248237751
1 segundos, buffer=100 bytes

Total bytes leidos: 248237751
0 segundos, buffer=1024 bytes
```

8.5 Archivos de texto

Como comentamos más arriba, el hecho de que un archivo sea considerado de texto o no tiene que ver con la forma en la que está codificada la información que contiene.

No es lo mismo un archivo que contiene los caracteres '1', '2', y '3' que un archivo que contiene los valores numéricos (de tipo `int`) 1, 2, 3. Veamos:

Binario	00110001	00110010	00110011
Decimal	49	50	51
Carácter	'1'	'2'	'3'

Fig. 8.7 Archivo de texto.

Binario	00000000	0000001	00000000	00000010	00000000	00000011
Decimal	1		2		3	
Carácter	'[]'	'[]'	'[]'	'[]'	'[]'	'[]'

Fig. 8.8 Archivo binario.

El primero se compone de una sucesión de *bytes* cuyos valores numéricos son 49, 50 y 51, que corresponden a los códigos ASCII de los caracteres ‘1’, ‘2’ y ‘3’. En cambio, en el segundo archivo tenemos grabados directamente los valores numéricos 1, 2 y 3, codificados cada uno en 16 bits.

El primer archivo podemos verlo, editarlo y modificarlo con cualquier editor de texto. Para manipular el contenido del segundo tendremos que utilizar algún programa especial que, probablemente, tendremos que desarrollar nosotros mismos.

La biblioteca estándar de C provee funciones con las cuales podemos leer y escribir caracteres y “líneas” en archivos de texto. Estas funciones son, entre otras: `getc`, `putc`, `fgets`, `fputs`, `fscanf` y `fprintf`.

8.5.1 Apertura de un archivo de texto

Para abrir un archivo de texto utilizamos la función `fopen` pasándole alguna de las siguientes cadenas como modalidad de apertura.

Modalidad de apertura	Descripción
"w+"	Crea un archivo sobre el que podemos grabar caracteres y líneas de texto y, leer y modificar caracteres y líneas ya grabadas. Si dentro de la carpeta existe un archivo con igual nombre entonces, al abrirlo, borrará todo su contenido dejándolo con 0 *bytes*.
"r+"	Idem anterior solo que si en la carpeta ya existe un archivo con igual nombre entonces no borrará su contenido. El indicador de posición del archivo apuntará al *byte* número 0.
"a+"	Idem anterior solo que el indicador de posición apuntará al final del archivo.

8.5.2 Leer y escribir caracteres (funciones `getc` y `putc`)

El siguiente programa abre dos archivos cuyos nombres recibe por línea de comandos. Luego lee caracteres del primero y los escribe en el segundo.

copiarArchivo.c

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   FILE* f1=fopen(argv[1],"r+");
   FILE* f2=fopen(argv[2],"w+");

   char c;
   while( (c=getc(f1))!=EOF )
   {
      putc(c,f2);
   }

   fclose(f2);
   fclose(f1);

   return 0;
}
```

8.5.3 Escribir líneas (función `fprintf`)

El siguiente programa lee las líneas de texto que ingresa el usuario por teclado y las graba en un archivo anteponiéndole el número de línea.

La función `gets` lee los caracteres que ingresa el usuario por el teclado hasta que presione [ENTER]. Todos estos caracteres conforman una "línea" de texto.

La función `fprintf` permite escribir datos formateados en un archivo de texto, usando las mismas máscaras que `printf`.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
   FILE* f1=fopen(argv[1],"w+");

   char linea[100];

   // muestro una flecha y luego leeo una linea por teclado
   printf("-->");
   gets(linea);

   int i=0;
   while( strcmp(linea,"FIN") )
   {

      // grabo el numero de linea y la linea en el archivo f1
      fprintf(f1,"%d, %s\n",i++,linea);

      // muestro una flecha y luego leeo una linea por teclado
      printf("-->");
      gets(linea);
   }

   fclose(f1);
}
```

8.5.4 Leer líneas (función `fgets`)

El siguiente programa lee líneas desde un archivo de texto cuyo nombre recibe en línea de comandos y luego, las muestra en la consola.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
   FILE* f1=fopen(argv[1],"r+");

   char linea[100];

   // leo una linea
   fgets(linea,100,f1);
```

```
   // mientras no llegue el fin del archivo
   while( !feof(f1)  )
   {
      printf("--> %s",linea);

      // leo la siguiente linea
      fgets(linea,100,f1);
   }

   fclose(f1);

   return 0;
}
```

8.5.5 Leer datos formateados (función `fscanf`)

La función `fscanf` (análoga a `scanf`) permite leer líneas con formato desde un archivo de texto. A continuación, veremos el archivo `texto.txt` y un programa que lee cada una de sus líneas e interpreta cada columna como “nombre”, “altura” y “edad”. Luego muestra estos datos por la consola.

```
texto.txt
Pablo     1.72 39
Juan      1.87 25
Pedro     1.65 32
Rolo      1.83 28
Cacho     1.55 30
```

```
#include <stdio.h>
#include <string.h>

int main()
{
   FILE* arch=fopen("texto.txt","r+");

   char nombre[11];
   float altura;
   int edad;

   fscanf(arch,"%s %f %d",nombre,&altura,&edad);
   while( !feof(arch) )
   {
      printf("%s, %.2f, %d\n",nombre,altura,edad);
      fscanf(arch,"%s %f %d",nombre,&altura,&edad);
   }

   fclose(arch);

   return 0;
}
```

La salida será:

```
Pablo, 1.72, 39
Juan, 1.87, 25
Pedro, 1.65, 32
Rolo, 1.83, 28
Cacho, 1.55, 30
```

8.6 Operaciones lógicas sobre archivos

En la primera parte del capítulo, estudiamos las operaciones primitivas de manejo de archivos. Abrir, cerrar, leer, escribir y cambiar, arbitrariamente, su indicador de posición para acceder directamente a cualquiera de sus registros o *bytes*.

Ahora nos concentraremos en el estudio de algoritmos que nos permitirán implementar operaciones lógicas a través de las cuales podremos manipular los registros de un archivo secuencial para ordenarlos, indexarlos, buscar e, incluso, "eliminar" registros.

8.6.1 Limitaciones de los archivos secuenciales

Como ya sabemos, un archivo secuencial se compone de una serie de *bytes* consecutivos almacenados en memoria secundaria. En este contexto, las únicas operaciones físicas que podemos realizar sobre el archivo son: lectura, escritura (o modificación), agregar información al final y manipular, arbitrariamente, el valor de su identificador de posición.

Por ejemplo, físicamente no podemos eliminar un registro del archivo. Podemos "mover la información", pero el tamaño del archivo seguirá siendo el mismo.

Imaginemos un archivo con 5 registros donde cada uno ocupa 10 *bytes*.

Registro 1	Registro 2	Registro 3	Registro 4	Registro 5	
0 … 9	10 … 19	20 … 29	30 … 39	40 … 49	*eof*

Para eliminar "Registro 3" tenemos que mover "Registro 4" y "Registro 5" a los *bytes* número 20 y 30 respectivamente. Veamos:

Registro 1	Registro 2	Registro 4	Registro 5	???	
0 … 9	10 … 19	20 … 29	30 … 39	40 … 49	*eof*

Esta operación, más allá del tiempo de procesamiento que podría utilizar, no tiene ningún sentido porque la longitud del archivo seguirá siendo la misma. El *eof* continúa ubicado luego del *byte* 49 y dejamos datos "inciertos" en la posición que, antes, ocupaba "Registro 5".

Por otro lado, si en lugar de eliminar a "Registro 3" eliminamos a "Registro 1" el tiempo de procesamiento será el mismo que utilizaría para reescribir todo el archivo menos, obviamente, un registro. Análogamente no podemos insertar un registro al principio o en el medio del archivo sin que esto implique un tiempo de procesamiento excesivo e inaceptable.

Para simplificar el estudio de estos temas, en todos los casos trabajaremos con el archivo `EMP.dat` cuya estructura de registro se describe a continuación:

emp.h

```
typedef struct Emp
{
   int idEmp;       // codigo de empleado
   char nom[30];    // nombre
   char dir[120];   // direccion
   long fecIngreso; // fecha de ingreso a la compania
}Emp;

// :
```

El lector ya sabe que la misma lógica será aplicable a archivos con registros de cualquier otro tipo de datos.

8.6.2 Ordenamiento de archivos en memoria

Por el momento, el único algoritmo de ordenamiento que hemos estudiado es el algoritmo de la burbuja que, ciertamente, es muy poco eficiente. Una implementación de este algoritmo aplicada al ordenamiento de archivos demandaría una cantidad de accesos (operaciones de entrada/salida) que sería inaceptable.

La única forma viable de ordenar un archivo mediante el uso de la burbuja es hacerlo en memoria, siempre y cuando la cantidad de registros que tenga el archivo sea acotada y razonablemente pequeña.

La implementación entonces consistirá en los siguientes tres pasos:

- Subir el archivo a memoria (`subirArchivo`)
- Ordenar su contenido (`ordenarContenido`)
- Bajar el archivo a disco (`bajarArchivo`)

Cuando hablamos de “subir el archivo a memoria” nos referimos a leer su contenido y almacenarlo en un *array* del mismo tipo de datos. Luego ordenamos el *array* y, por último, “bajamos el archivo” generando uno nuevo o reescribiendo el contenido del archivo original.

Veamos los diagramas de cada una de estas acciones:

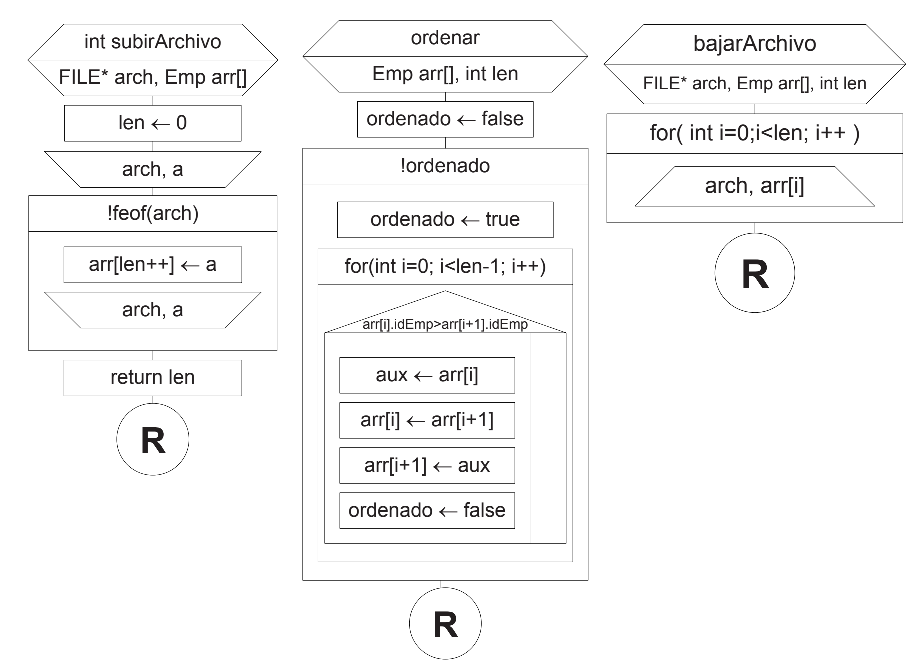


Fig. 8.9 Operaciones para subir, ordenar y bajar archivos.

La función `subirArchivo` retorna la longitud (*len*) del *array* en el que “subimos” todos los registros del archivo.

La función `ordenar` es exactamente la misma que estudiamos en el Capítulo 7, pero, claro, comparando el campo `idEmp` del registro del *array*.

Respecto de `bajarArchivo`, simplemente recorremos el *array* y grabamos cada uno de sus registros en el archivo que recibimos como parámetro.

Ahora, el siguiente programa abre un archivo, lo sube a memoria, ordena su contenido y lo reescribe ordenado.

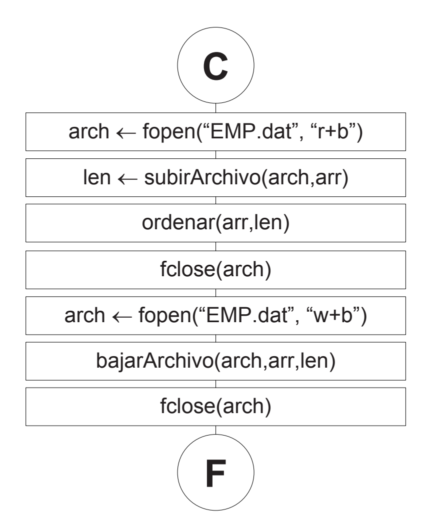


Fig. 8.10 Ordena un archivo en memoria.

Veamos el código fuente del programa:

testOrdenaArchivo.c

```
#include <stdio.h>
#include "emp.h"

int main()
{
   Emp arr[100];
   FILE* f;

   // abro el archivo, lo subo, lo ordeno y lo cierro
   f=fopen("EMP.dat","r+b");
   int len=subirArchivo(f,arr);
   ordenar(arr,len);
   fclose(f);

   // lo abro para escritura, lo bajo y lo cierro
   f=fopen("EMP.dat","w+b");
   bajarArchivo(f,arr,len);
   fclose(f);

   return 0;
}
```

Veamos la codificación de las funciones:

emp.c

```
// :

// prototipos, se desarrollan mas abajo
void leerEmp(FILE*, Emp*);
void grabarEmp(FILE*, Emp*);

// recorre el archivo (que viene abierto) y
// asigna cada registro en arr
int subirArchivo(FILE* arch, Emp arr[])
{
   Emp reg;
   int len=0;
   leerInt(arch,&reg);

   while( !feof(arch) )
   {
      arr[len++]=reg;
      leerEmp(arch,&reg);
   }

   return len;
}

// ordena el array mediante el algoritmo de la burbuja
void ordenar(Emp arr[],int len)
{
   int ordenado=0;
   while( !ordenado )
   {
      ordenado=1;
      for(int i=0; i<len-1; i++)
      {
         if( arr[i].idEmp>arr[i+1].idEmp )
         {
            int aux=arr[i];
            arr[i]=arr[i+1];
            arr[i+1]=aux;
            ordenado=0;
         }
      }
   }
}

// recorre el array y graba elemento en el archivo
void bajarArchivo(FILE* arch, Emp arr[], int len)
{
   for(int i=0; i<len; i++ )
   {
      grabarEmp(arch,arr+i);
   }
}
```

```
// lee un entero desde el archivo
void leerEmp(FILE* a, Emp* reg)
{
   fread(reg,sizeof(Emp),1,a);
}

// escribe un entero en el archivo
void grabarEmp(FILE* a, Emp* reg)
{
   fwrite(reg,sizeof(Emp),1,a);
}
```

8.6.3 Relación entre el número de byte y el número de registro

Genéricamente hablando, si *n* es la posición del último *byte* de un archivo de registros de tipo *T* cuya longitud de tipo es `sizeof(T)`, podemos asegurar que el archivo tiene exactamente *n*/`sizeof(T)` registros.

Desde el punto de vista lógico, probablemente nos interese más hablar del número de registro que del número de *byte*. Así que desarrollaremos dos funciones extremadamente simples que nos permitirán, dado un número de *byte* obtener el número de registro al que apunta y, a la inversa, dado un número de registro obtener el número de *byte* en donde comienza. Las llamaremos `byteToRecno` y `recnoToByte`.

emp.c

```
// :
long byteToRecno(int byteNo)
{
    return byteNo/sizeof(Emp);
}

long recnoToByte(int recNo)
{
    return recNo*sizeof(Emp);
}
```

Estas funciones *hardcodean* el tipo de datos `Emp` por lo que, si trabajamos con archivos que contengan registros de otro tipo tendremos que modificarlas o desarrollar otro par de funciones semejantes.



hardcodear: incrustar datos en el código en lugar de obtenerlo de una fuente externa.

8.6.4 Búsqueda binaria sobre archivos

Si el archivo está ordenado entonces podemos utilizar el algoritmo de la búsqueda binaria para buscar un registro que contenga un determinado valor.

Aunque la lógica del algoritmo es la misma que estudiamos en el Capítulo 7, su implementación deberá sufrir algunas modificaciones.

Veamos el diagrama y luego lo analizaremos.

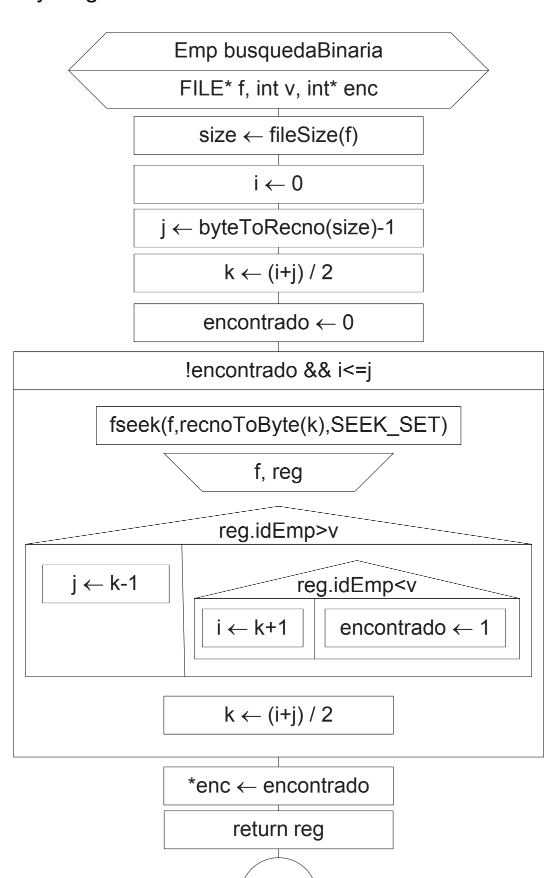


Fig. 8.11 Búsqueda binaria sobre un archivo.

Sabemos que el algoritmo de la búsqueda binaria funciona con dos índices, `i` y `j`, que deben apuntar respectivamente al primero y al último registro del archivo. Es decir que si el archivo tiene 100 registros entonces `i` debe ser 0 y `j` debe ser 99. Para obtener este último valor, utilizamos las funciones `fileSize` y `byteToRecno` analizadas más arriba.

Luego calculamos `k` como el número de registro promedio entre `i` y `j` y accedemos al archivo para leerlo. Para esto, ubicamos el indicador de posición en el *byte* correspondiente que obtenemos invocando a la función `recnoToByte`.

La función `búsquedaBinaria` asigna *true* o *false* a `enc` según exista o no el registro que estamos buscando. En caso afirmativo, su valor de retorno será el contenido del registro encontrado.

8.6.5 Indexación

Supongamos que tenemos un libro cuyos capítulos se presentan en orden según un criterio didáctico y lógico dado por el grado de complejidad del tema que tratan.

Por ejemplo:

Capítulo	Título	Página
1	Introducción a la programación	1
2	Operadores lógicos	34
3	Funciones y metodología *top-down*	46
4	Tipos de datos alfanuméricos	68
5	Punteros a carácter	123
6	Punteros, *arrays* y código compacto	150
7	Tipos de datos estructurados	179
8	Operaciones sobre archivos	280

El orden de los capítulos es el adecuado para un lector (quizás principiante) interesado en aprender todos estos temas. Sin embargo, si el lector quiere consultar por un tema en particular sería ilógico pretender que recorra, secuencialmente, las páginas del libro hasta encontrar lo que le interesa leer.

Por este motivo, los libros traen un índice alfabético que implementa un mecanismo de acceso directo a la información. En nuestro caso, el índice podría ser el siguiente:

Título	Capítulo	Página
Funciones y metodología *top-down*	3	46
Introducción a la programación	1	1
Operaciones sobre archivos	8	280
Operadores lógicos	2	34
Punteros a carácter	5	123
Punteros, *arrays* y código compacto	6	150
Tipos de datos alfanuméricos	4	68
Tipos de datos estructurados	7	179

Si el lector está interesado en consultar puntualmente el tema "Punteros a carácter" lo buscará, alfabéticamente, en el índice y, al encontrarlo, obtendrá el número de la página en donde el tema se comienza a tratar.

Físicamente, el orden de los capítulos del libro sigue siendo el mismo, pero el lector puede acceder directamente al contenido que le interesa leer, previa búsqueda en el índice. Este mecanismo le ahorra la tediosa tarea de recorrer, página por página, hasta llegar al tema en el cual está interesado.

8.6.6 Indexación de archivos

Como explicamos anteriormente, en general, ordenar un archivo no es una buena idea. Sin embargo, podemos indexarlo y, de esta forma, acceder "en orden" a su contenido.

Indexar el archivo implica crear una tabla (*array* de estructuras) con dos campos que llamaremos "clave" y `posicion`. El primero será el valor por el cual queremos ordenar el contenido el archivo. El segundo será la posición que, dentro del archivo, ocupa el registro que contiene dicho valor clave.

Volviendo al archivo `EMP.dat`, supongamos que su contenido es el siguiente:

	`idEmp`	`nom`	`dir`	`fecIngreso`
0	40	Juan	Los Alamos 234	20061011
1	20	Pedro	El Tala 2331	20081109
2	30	Carlos	Av. San Jose 3116	20070807
3	10	Rolo	Pje. Elefante 12	20050815
4	60	Julio	Av. de los Incas 43	20100623
5	50	Diego	Los Mirasoles 76	20090122

eof

Entonces, la siguiente tabla lo indexa por número de empleado (`idEmp`):

	idEmp	posicion
0	10	3
1	20	1
2	30	2
3	40	0
4	50	5
5	60	4

La tabla está ordenada por el campo clave `idEmp` y, asociado a cada clave tenemos la posición que, dentro del archivo, ocupa el registro que contiene dicho valor. Luego, si recorremos la tabla y por cada fila utilizamos el campo `posicion` para posicionarnos en el archivo accederemos a cada uno de sus registros en orden ascendente según su valor de `idEmp`.

Definamos, entonces, la estructura `IdxIdEmp` con los campos `idEmp` y `posicion` y desarrollemos la función `indexar`.

emp.h

```
// :
typedef struct IdxIdEmp
{
   int idEmp;
   int posicion;
}IdxIdEmp;
```

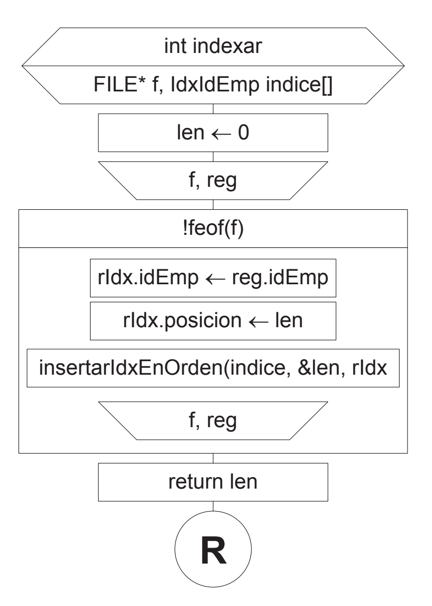


Fig. 8.12 Algoritmo de la función indexar.

En la función `indexar`, invocamos a la función `insertarIdxEnOrden` que no es más que una implementación particular de la función `insertarEnOrden` que estudiamos en el Capítulo 7. Esta función inserta un registro de tipo `IdxIdEmp` en el *array* `indice`.

A su vez, `insertarIdxEnOrden` invoca a `insertarIdx`, su propia implementación de la función `insertar`.

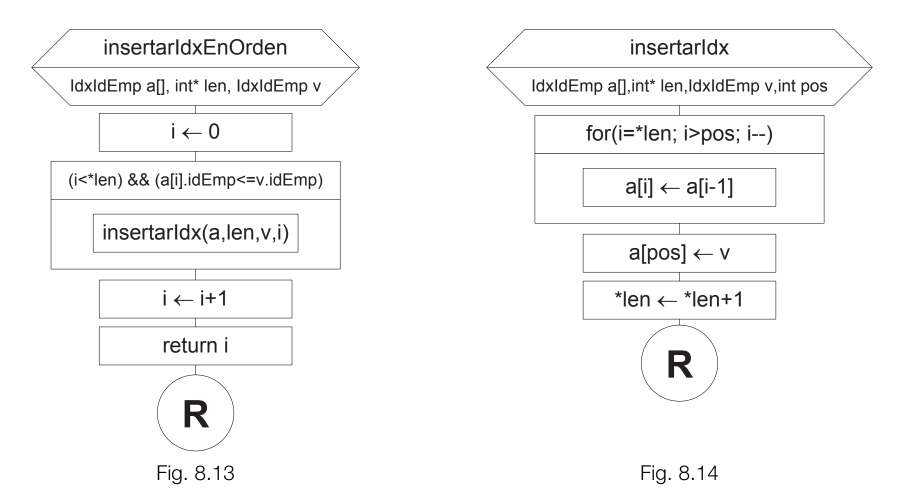


Fig. 8.13

Fig. 8.14

Veamos la codificación de estas funciones:

emp.c

```
//:
void insertarIdx(IdxIdemp a[],int* len, IdxIdEmp v,int pos)
{
   for(int i=*len; i>pos; i--)
   {
      a[i]=a[i-1];
   }

   a[pos]=v;
   *len=*len+1;
}

int insertarIdxEnOrden(IdxIdEmp a[], int* len, IdxIdEmp v)
{
   int i=0;
   while(i<*len && a[i].idEmp<=v.idEmp)
   {
      i=i+1;
   }

   insertarIdx(a,len,v,i);
   return i;
}

int indexar(FILE* f, IdxIdEmp indice[])
{
   int len=0;
   IdxIdEmp rIdx;

   int reg;
   leerInt(f,&reg);
```

```
   while( !feof(f) )
   {
      rIdx.idEmp=reg.idEmp;
      rIdx.posicion=len;
      insertarIdxEnOrden(indice,&len,rIdx);
      leerInt(f,&reg);
   }

   return len;
}
```

Ahora analicemos un programa que, luego de indexar el archivo `EMP.dat`, accede y muestra su contenido en orden.

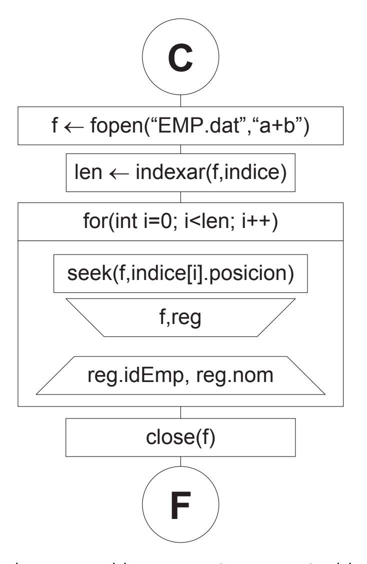


Fig. 8.15 Indexa un archivo y muestra su contenido en orden.

La salida de este programa, considerando los datos del archivo `EMP.dat` analizado más arriba, será la siguiente:

```
10    Rolo
20    Pedro
30    Carlos
40    Juan
50    Diego
60    Julio
```

Veamos su codificación:

testIndexar.c

```
#include <stdio.h>
#include "emp.h"

int main()
{
   FILE* f=fopen("EMP.dat","a+b");
   IdxIdEmp indice[100];

   // indexo el archivo
   int len = indexar(f,indice);

   Emp reg;
   for(int i=0; i<len; i++)
   {
      fseek(f,indice[i].posicion*sizeof(Emp),SEEK_SET);
      leerEmp(f,&reg);

      printf("%d\n",reg);
   }

   fclose(f);

   return 0;
}
```

En el ejemplo anterior, indexamos el archivo `EMP.dat` para "ordenar" su contenido según el campo `idEmp`. ¿Qué cambios y/o modificaciones habría que aplicar para que la indexación se realice según los valores del campo `nom`? Este análisis y posterior desarrollo queda por cuenta y cargo del lector.

8.6.7 Eliminar registros en un archivo (bajas lógicas)

Como ya explicamos, físicamente resulta imposible eliminar registros de un archivo secuencial. Sin embargo, desde el punto de vista lógico existen varias estrategias de solución para este problema. Recordemos la estructura de los registros del archivo `EMP.dat`.

```
typedef struct Emp
{
   int idEmp;       // codigo de empleado
   char nom[30];    // nombre
   char dir[120];   // direccion
   long fecIngreso; // fecha de ingreso a la compania
}Emp;
```

Supongamos que el archivo tiene el siguiente contenido:

	idEmp	nom	dir	fecIngreso
0	40	Juan	Los Alamos 234	20061011
1	20	Pedro	El Tala 2331	20081109
2	30	Carlos	Av. San Jose 3116	20070807
3	10	Rolo	Pje. Elefante 12	20050815
4	60	Julio	Av. de los Incas 43	20100623
5	50	Diego	Los Mirasoles 76	20090122

eof

La estrategia más simple consiste en "marcar" el registro que queremos eliminar para, luego, simplemente ignorarlo.

Por ejemplo, si decidimos eliminar el registro:

2	30	Carlos	Av. San Jose 3116	20070807

La "marca" de baja lógica podría consistir en multiplicar por -1 el valor del campo `idEmp` que, en este caso, quedaría así:

2	-30	Carlos	Av. San Jose 3116	20070807



Fabio Zerpa
Actor de profesión, se interesó por la ufología. A comienzos de los años sesenta, comenzó a dar sus primeras conferencias. Condujo el programa televisivo "La Casa Infinito" que salía al aire para toda Latinoamérica por el canal Infinito.

Luego deberíamos ignorar a todos aquellos registros cuyo valor de `idEmp` sea negativo.

Esta estrategia que resulta tan fácil de implementar, en un contexto real resultaría inaplicable.

"No estamos solos en el universo" dijo Fabio Zerpa. Esto, extrapolado al contexto de la programación de aplicaciones significa que un mismo archivo, muy probablemente, será utilizado por más de un programa y/o sistema. Por esta razón, si modificamos, unilateralmente, su contenido alguna otra aplicación podría dejar de funcionar o (peor aún) podría trabajar con datos inconsistentes.

Por ejemplo, supongamos que la empresa en donde trabajan los empleados cuyos datos contiene el archivo `EMP.dat` tiene una aplicación que registra la actividad que realizan estos empleados durante las horas de trabajo. Este registro se lleva a cabo agregando filas en el archivo `HORAS.dat` cuya estructura de registro vemos a continuación:

```
typedef struct Hora
{
   int idEmp;              // codigo de empleado
   long fecha;             // fecha
   char descripcion[200];  // tarea que desarrollo
   int cantHoras;          // tiempo que le dedico a la tarea
}Hora;
```

Es decir, al finalizar cada día de trabajo los empleados de la empresa agregan registros en este archivo con el detalle de las actividades que realizaron durante la jornada laboral.

Un ejemplo de los datos que este archivo podría tener es el siguiente:

	idEmp	fecha	nom	cantHoras
0	20	20110305	Documentación...	3
1	20	20110305	Programación...	5
2	30	20110305	Investigación...	2
3	30	20110305	Depuración...	2
4	30	20110305	Chateando...	6
:	:	:	:	:

eof

Según estos registros el empleado cuyo `idEmp` es 20 (Pedro), el día 5/3/2011 trabajó 3 horas documentando y 5 horas programando. Y el empleado cuyo `idEmp` es 30 (Carlos), el mismo día, trabajó 2 horas investigando y luego se pasó 6 horas chateando.

Los archivos `HORAS.dat` y `EMP.dat` están relacionados a través del campo `idEmp`. Este valor (llegado el caso) le permitirá a la aplicación que registra la actividad de los empleados mostrar los datos personales de cada uno. Si en `EMP.dat` modificamos el valor de `idEmp` de Carlos, el archivo `HORAS.dat` quedará inconsistente. Es decir, tendrá registros cuyo `idEmp` no se corresponda con ningún registro de `EMP.dat`.

Si bien este análisis excede el alcance de un curso de algoritmos y, más bien, corresponde a un curso de diseño de sistemas, no está demás que el lector tenga presente que existen situaciones de este tipo.

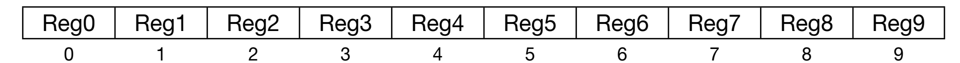
En una implementación real, los archivos `EMP.dat` y `HORAS.dat` serían tablas de una base de datos relacional con restricciones de integridad. La misma base de datos no permitirá modificar el valor de `idEmp` en la tabla `EMP` porque esta acción dejaría registros "huérfanos" en la tabla `HORAS`.

8.6.8 Bajas lógicas con soporte en un archivo auxiliar

Para no alterar los datos de los registros que "eliminamos" tenemos que crear un archivo auxiliar y grabar allí la información sobre las bajas.

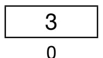
Por ejemplo, el siguiente archivo tiene 10 registros que podemos identificar por su posición.

`ARCHIVO.dat`

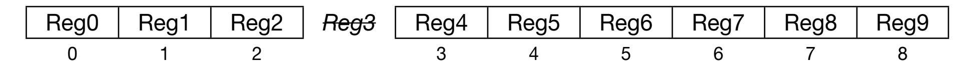


Para dar de baja al registro ubicado en la posición número 3 grabaremos este valor en un nuevo archivo. Lo llamaremos `BAJAS.dat`

`BAJAS.dat`



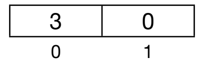
`ARCHIVO.dat`



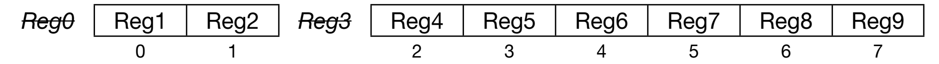
Físicamente, "Reg3" sigue ubicado en el cuarto lugar, pero, desde el punto de vista lógico, todos los registros posteriores a esta posición "bajaron" un lugar.

Si ahora damos de baja el registro número 0 todos los registros ubicados en las posiciones posteriores "bajarán" un lugar.

`BAJAS.dat`

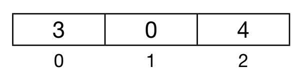


`ARCHIVO.dat`

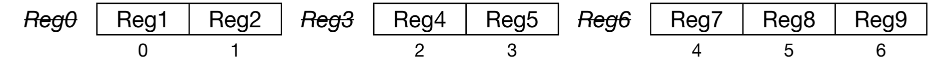


Por último, demos de baja al registro número 4.

`BAJAS.dat`



`ARCHIVO.dat`



Como vemos, la estrategia es muy simple. Sin embargo, su implementación puede presentar algunas complicaciones interesantes por lo que la estudiaremos en detalle en el siguiente capítulo.

8.7 Resumen

En este capítulo, estudiamos el concepto de archivo. Vimos que, según su contenido, el archivo puede considerarse como archivo binario, archivo de texto o archivo de registros.

Estudiamos las operaciones físicas y primitivas que nos provee el lenguaje de programación: abrir, leer, escribir, posicionar y cerrar el archivo.

Estudiamos también un conjunto de operaciones lógicas gracias a las cuales pudimos ordenar archivos, buscar registros, indexarlos y (aunque no lo implementamos aún) eliminar registros mediante el concepto de “baja lógica”.

En el próximo capítulo, veremos cómo desarrollar “tipos de datos abstractos” que nos permitan encapsular la lógica de ciertas implementaciones algorítmicas complejas y desarrollaremos un TAD (tipo de datos abstracto) para encapsular la implementación de “baja lógica con soporte en un archivo auxiliar” que analizamos más arriba.

8.8 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

8.8.1 Mapa conceptual

8.8.2 Autoevaluaciones

8.8.3 Videotutoriales

8.8.3.1 Leer y escribir un archivo

8.8.3.2 Leer y escribir un archivo de registros

8.8.4 Presentaciones*

9

Tipo Abstracto de Dato (TAD)

Contenido

Objetivos del capítulo

- Analizar y descubrir la existencia de diferentes capas de abstracción.
- Comprender el concepto de "tipo de dato abstracto" (TAD)
- Analizar los tipos `FILE`, `FILE*` y la operaciones asociadas: `fopen`, `fread`, `fwrite`, etcétera.
- Lograr abstracción mediante el ocultamiento de la lógica algorítmica.
- Desarrollar el TAD `Fecha`, simple y como introducción al tema.
- Desarrollar el TAD `XFile` para encapsular la técnica de "baja lógica" sobre los registros de un archivo.

Competencias específicas

- Representar y aplicar los tipos de datos abstractos por medio de un lenguaje de programación C.



En la Web de apoyo del libro encontrará el contenido de este capítulo para leer online:

1. Ir a la pagina http://virtual.alfaomega.com.mx
2. Ingresar los nombres de Usuario y Contraseña definidos cuando registro el libro (ver Registro en la Web de apoyo).
3. Hacer un clic en el hipervínculo que lleva el nombre del capítulo que desea leer.

10

Análisis de ejercicios integradores

Contenido

Objetivos del capítulo

- Desarrollar algoritmos completos, que requieran aplicar los conocimientos adquiridos durante los capítulos anteriores.
- Analizar y resolver problemas de corte de control y de apareo de archivos.

Competencias específicas

- Aplicar los conocimientos adquiridos de programación estructurada en un programa con algún nivel de complejidad.

10.1 Introducción

Llegamos a un punto en el cual contamos con todos los recursos necesarios como para encarar y resolver problemas con cierto nivel de complejidad.

Entre los Capítulos 1 y 6, explicamos el lenguaje de programación, el uso de variables, funciones, argumentos por valor y referencia y la relación que, en C, existe entre los punteros y los *arrays*.

En el Capítulo 7, estudiamos operaciones sobre *arrays* y tipos de datos estructurados y, en el Capítulo 8, operaciones sobre archivos.

En este capítulo, analizaremos problemas completos, cuya resolución requerirá utilizar todos los conocimientos adquiridos anteriormente.

10.2 Problemas con corte de control

Cuando trabajamos con archivos cuyos registros están ordenados por un determinado criterio, podemos identificar subconjuntos o grupos de registros que responden a una característica común.

Veamos los siguientes ejemplos:

`ALUMNOS.dat`

matricula	**nombre**	**anioIngr**
1442	Juan	2005
:	:	:
1567	Carlos	2005
1785	Matias	2006
:	:	:
1886	Alfredo	2006
1890	Tobias	2007
:	:	:

`INFRACCIONES.dat`

placa	**fecha**	**descrip**
AQN123	0723	Pasa con luz roja
:	:	:
AQN123	0912	Excede velocidad
CGE544	0615	Mal estacionado
:	:	:
CGE544	0816	Alcohol en sangre
:	:	:

El archivo `ALUMNOS.dat` tiene información de los alumnos de una universidad y sus registros se encuentran ordenados por el campo `anioIngr` (año de ingreso). Como vemos, todos los alumnos que ingresaron en el año 2005 se encuentran agrupados, luego todos los que ingresaron en el 2006, etcétera.

Análogamente, en el archivo `INFRACCIONES.dat`, ordenado por el campo `placa`, vemos agrupadas todas las infracciones del vehículo cuya placa (o chapa patente) es AQN123, luego todas las de CGE544, etcétera.

Cuando hablamos "corte de control" nos referimos a recorrer, secuencialmente, el archivo "cortando" cada vez que terminamos de procesar un conjunto de registros para proporcionar información referida al grupo de registros que acabamos de procesar y luego reprocesarla para elaborar más información, en este caso, refiera a la totalidad de los registros del archivo.

Problema 10.1

Una tienda registra el detalle de sus ventas en el archivo `VTASDET.dat` cuya estructura de registro es la siguiente.

```
typedef struct VtaDet
{
   int nroTk;     // numero de ticket o factura
   int idArt;     // codigo de articulo
   int cant;      // cantidad de unidades vendidas
   int cantEntr;  // cantidad de unidades efectivamente entregadas
}VtaDet;
```

El archivo está ordenado por el campo `nroTk` de forma tal que todos los registros con igual número de *ticket* se encuentran agrupados.

Se pide emitir el siguiente listado:

Número de Ticket	*Cantidad de items*	*Cantidad promedio de unidades por item*	*Porcentaje de artículos entregados*
9999	99	99	99
9999	99	99	99
:	:	:	:

Cantidad total de tickets procesados: 99
Cantidad tickets 100% entregados: 99

Análisis

Sabemos que el archivo está ordenado por el campo `nroTk`. Veamos un ejemplo del contenido que podría llegar a tener.

nroTk	**idArt**	**cant**	**cantEntr**
1	5	1	1
1	2	3	2
1	4	1	1
1	8	2	0
2	4	1	0
2	1	5	3
2	2	3	1
3	8	1	1
3	4	2	1
4	3	1	1
4	1	2	0
4	5	2	1
5	3	1	1
5	1	3	1
5	5	2	2

En el archivo, podemos identificar conjuntos de registros con el mismo valor de `nroTk` y, además, como el archivo está ordenado por dicho campo vemos que todos estos registros se encuentran agrupados.

Cada *ticket* (`nroTk`) representa una compra compuesta por diferentes cantidades de 1 o más artículos. En el ejemplo, la compra cuyo `nroTk` es 3 se compone 1 unidad del artículo 8 y 2 unidades del artículo 4, una de las cuales aún no fue entregada.

Luego, si recorremos secuencialmente el archivo controlando que cada registro leído tenga el mismo valor de `nroTk` que el anterior, al finalizar el proceso de cada grupo de registros podremos emitir las filas del listado solicitado, detallando:

- El número de *ticket* procesado.
- La cantidad de ítems (artículos diferentes) que componen la compra.
- La cantidad promedio de unidades por ítem.
- El porcentaje de artículos entregados.

En cambio, para emitir los resultados requeridos al pie del listado:

- Cantidad de *tickets* procesados.
- Cantidad de *tickets* 100% entregados.

será necesario procesar la totalidad de los registros del archivo.

En otras palabras, para conocer la cantidad de ítems (o artículos diferentes) que componen la compra cuyo `nroTk` es 1 bastará con incrementar un contador (`cantItem`) cada vez que leamos un registro con dicho valor en el campo `nroTk`. Además, si sumamos la cantidad de unidades de cada artículo (`sumaCant`) podremos obtener el promedio de unidades adquiridas dividiendo esta suma por la cantidad anterior. Y, por último, si sumamos las cantidades entregadas (`sumaEntr`) podremos estimar el porcentaje de artículos entregados como: `sumaEntr*100/sumaCant`.

En cambio, para obtener la cantidad total de *tickets* procesados tendremos que incrementar un contador (`cantTk`) cada vez que hayamos finalizado el proceso de todos los ítems que componen un mismo *ticket*. Y para conocer la cantidad de *tickets* cuyos artículos fueron completamente entregados incrementaremos un contador (`cant100`) solo si `sumaCant` es igual a `sumaEntr`. Esta información recién estará completa y podrá emitirse cuando hayamos procesado la totalidad de los registros del archivo.

A los problemas, en los cuales el algoritmo de solución implica recorrer secuencialmente un archivo controlando que no cambie el valor de un determinado campo, los llamamos problemas "con corte de control".

Notemos que es fundamental el hecho de que el archivo se encuentre ordenado. De lo contrario, el corte de control no sería aplicable.

El diagrama de la Fig. 10.1 muestra como recorrer el archivo `VTASDET.dat` "cortando" cada vez que cambia el valor de `nroTk`. Ver Fig. 10.1.

Como comentamos más arriba, la idea es recorrer secuencialmente el archivo "cortando" cada vez que leamos un registro cuyo valor de control no coincida con el valor del registro anterior. En nuestro caso, el valor de control es `nroTk`.

El problema se resuelve anidando dos ciclos de repetición. El primero iterará mientras que no llegue el fin del archivo. El segundo, además, iterará mientras que no cambie el valor que identifica al grupo de registros que estamos procesando (`nroTk`). Cuando leamos un registro cuyo `nroTk` sea diferente del anterior (`nroTkAnt`) el ciclo interno dejará de iterar, pero luego, el ciclo externo forzará a que ese registro se procese como parte del siguiente grupo.

Los dos ciclos anidados definen 5 secciones (numeradas de 1 a 5 en el diagrama anterior).

A la sección (1) solo ingresan registros que tienen el mismo `nroTk`. En nuestro ejemplo, el ciclo interno iterará tantas veces como ítems tenga el *ticket* o factura que estamos procesando. Es decir que si aquí incrementamos un contador (`cantItem`) y acumulamos las cantidades solicitadas (`sumaCant`) y las cantidades entregadas (`sumaEntr`) entonces, en la sección (2), podremos mostrar la cantidad de ítems que componen a la factura, la cantidad promedio de unidades por ítem y el porcentaje entregado. Estas variables deben volver a cero antes de comenzar el proceso de los ítems del siguiente *ticket* por lo tanto, en (3), las tendremos que inicializar.

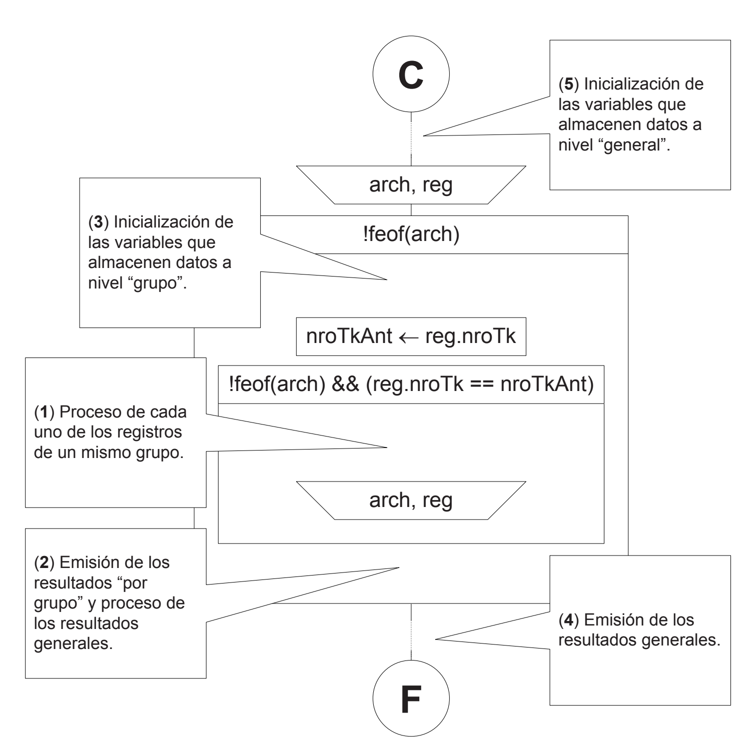


Fig. 10.1 Esquema de corte de control por 1 variable.

A la sección (2) llegamos luego de procesar la totalidad de los ítems de cada *ticket*. Si aquí incrementamos un contador (`cantTk`), al finalizar el proceso (esto es en 4), podremos informar la cantidad total de *tickets* procesados. Además, si aquí verificamos que `sumaCant` sea igual a `sumaEntr` e incrementamos un contador (`cant100`), luego, también en (4), podremos mostrar la cantidad de *tickets* cuyos artículos fueron completamente entregados.

El algoritmo pasará por la sección (3) justo antes de comenzar a procesar cada grupo de registros con igual valor en `nroTk`, por lo tanto, este es el lugar adecuado para inicializar las variables relacionadas a cada *ticket*. (`cantItem`, `sumaCant` y `sumaEntr`). En cambio, por la sección (5) se pasará una única vez al comienzo del programa.

La sección (5) es el lugar adecuado para dar valor inicial a las variables cuyos datos se relacionan con la totalidad de los *tickets* o facturas: `cantTk` y `cant100`.

Veamos el diagrama del programa principal:

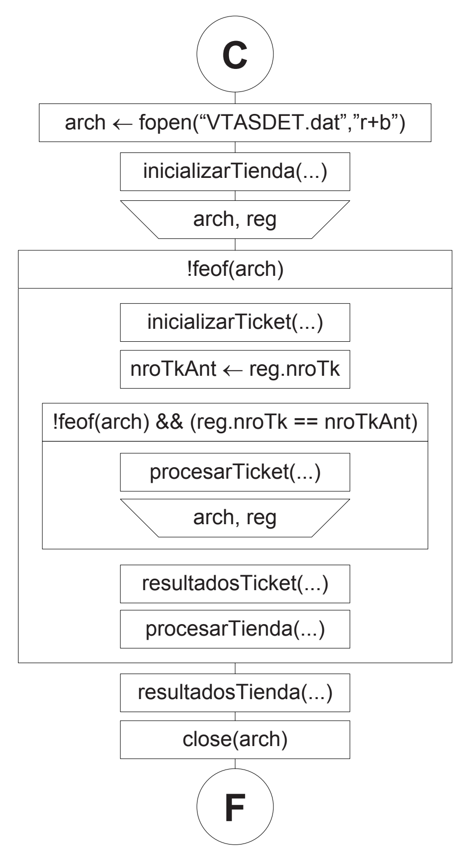


Fig. 10.2 Programa principal.

Como vemos, resolvimos el programa principal invocando a módulos que describen la tarea que debemos realizar en cada sección.

Al módulo `procesarTicket` solo ingresarán registros con el mismo valor de `nroTk`. Si sumamos las cantidades solicitadas (`sumaCant`) y las cantidades entregadas (`sumaEntr`) e incrementamos un contador de ítems (`cantItem`), cuando lleguemos a `resultadosTicket` podremos mostrar, por cada *ticket*, la cantidad de ítems que lo componen, la cantidad de unidades promedio y el porcentaje de entrega cumplido.

Todas estas variables deben estar inicializadas antes de procesar los ítems de cada *ticket*. Esto lo haremos en `inicializarTicket`.

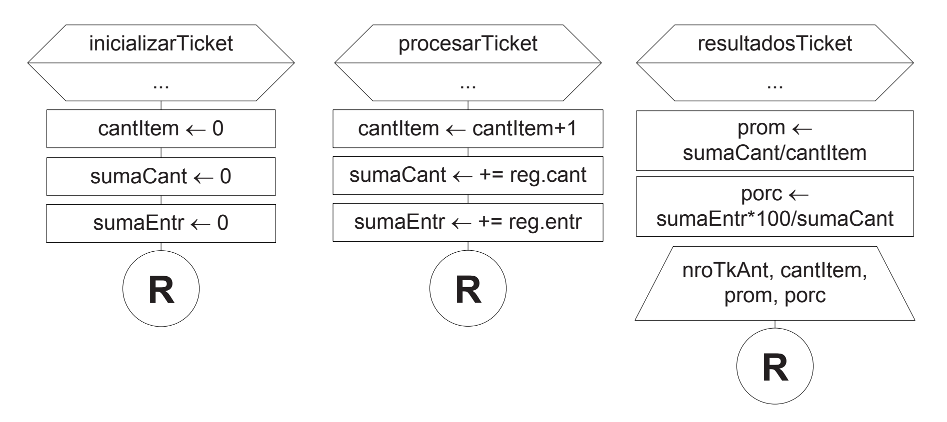


Fig. 10.3 Módulos que procesan la información relacionada a "cada *ticket*".

Notemos que para mostrar el número de *ticket* en `resultadosTicket` tuvimos que utilizar la variable `nroTkAnt`. Esto se debe a que, a esta altura, los datos que contiene `reg` corresponden al siguiente grupo de registros, pero la variable `nroTkAnt` todavía mantiene el número del *ticket* anterior.

Luego de mostrar la información relacionada al *ticket* que acabamos de procesar tenemos que trabajar con los datos relacionados a la facturación total de la tienda. Esto lo hacemos en `procesarTienda`.

Al módulo `procesarTienda` llegaremos cada vez que terminemos de procesar los ítems de cada *ticket*. Si aquí incrementamos un contador (`cantTk`) podremos informar, en `resultadosTienda`, la cantidad total de *tickets* procesados. Y, si se verifica que `sumaCant` es igual que `sumaEntr`, otro contador (`cant100`) nos permitirá mostrar la cantidad de *tickets* completamente entregados.

Estas variables mantienen datos relacionados con todos los *tickets* detallados en el archivo, por lo tanto, deben inicializarse una única vez, en `inicializarTienda`.

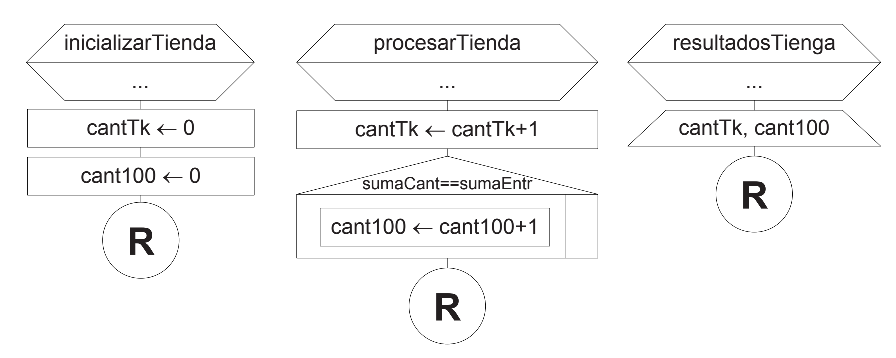


Fig. 10.4 Módulos relacionados con el proceso de la información "general" o "de la tienda".

Por cuestiones de simplicidad, preferí omitir la lista de argumentos que se le deben pasar a cada módulo que, por cierto, sería demasiado extensa. En un diagrama, esto es aceptable siempre y cuando los nombres de variables que utilicemos dentro del módulo coincidan con los nombres de variables utilizados en el programa principal desde el cual el módulo fue invocado.

Recordemos que en C no existen los parámetros por referencia, por lo tanto, una implementación real de este algoritmo exigirá trabajar con punteros.

10.2.1 Archivos de novedades vs. archivos maestros

En el problema anterior, procesamos un archivo cuyos registros representan las ventas realizadas por una tienda durante un determinado período de tiempo.

El archivo `VTASDET.dat` es dinámico. Cada vez que la tienda efectúe una nueva venta agregará uno o más registros al final del archivo y, como el objetivo de la tienda es vender, `VTASDET.dat` constantemente estará recibiendo nuevos registros.

Desde el punto de vista de las ventas, `VTASDET.dat` contiene las novedades que acontecen en la tienda ya que cada nueva venta aparecerá registrada al final del archivo.

Pensemos ahora en el archivo `ARTICULOS.dat` cuya estructura de registro es la siguiente:

```
typedef struct Articulo
{
   int idArt;      // codigo de articulo
   char desc[50];  // descripcion del articulo
   double precio;  // precio de venta
   int idRubro;    // codigo de rubro al que pertenece
}Articulo;
```

Cada registro de este archivo representa un artículo de los que la tienda comercializa.

La información contenida en `ARTICULOS.dat` es estática si la comparamos con la que contiene `VTASDET.dat`. Es decir, los artículos pueden cambiar de precio de venta e, incluso, pueden incorporarse nuevos artículos para su posterior comercialización, pero todo esto es esporádico y eventual.

`ARTICULOS.dat` es una especie de catálogo de artículos que llamaremos “archivo maestro”, a cuyos registros vamos a acceder cada vez que necesitemos obtener la descripción de un artículo, su precio de venta, etcétera.

En general, trabajaremos con un archivo de novedades (o movimientos) y varios archivos maestros (o de consulta) cuyos registros proveerán información complementaria para los registros de novedades.

Problema 10.2

En el mismo contexto del problema anterior, se cuenta con los siguientes archivos:

`VTASDET.dat`

```
typedef struct VtaDet
{
   int nroTk;
   int idArt;
   int cant;
   int cantEntr;
}VtaDet;
```

`ARTICULOS.dat`

```
typedef struct Articulo
{
   int idArt;
   char desc[50];
   double precio;
   int idRubro;
}Articulo;
```

`VTASDET.dat` está ordenado por `nroTk`. `ARTICULOS.dat` está ordenado por `idArt`.

Se pide emitir el siguiente listado:

Número de Ticket: 9999

Id. Artículo	*Descripción*	*Precio de Venta*	*Cantidad*	*Monto*
9999	xxxxxxxx	99.99	99	99.99
9999	xxxxxxxx	99.99	99	99.99
:	:	:	:	:
			Total Ticket:	999.99

Cantidad total de tickets procesados:	99
Monto total facturado	99.99
Monto promedio facturado	99.99

Análisis

El listado se compone de dos partes principales. La primera aparecerá tantas veces como *tickets* existan en `VTASDET.dat`. La segunda aparecerá una única vez, luego de procesar todos los registros del archivo.

Es decir, por cada *ticket* tenemos que emitir un listado que se compone de una cabecera que indica el número de *ticket*, uno o varios ítems con el detalle de cada uno de los productos adquiridos y un pie que indica el importe total del *ticket*.

Si cada vez que finaliza el proceso de un *ticket* incrementamos un contador y reacumulamos los totales podremos luego, en `resultadosTienda`, mostrar la segunda parte del listado.

El problema se resuelve recorriendo `VTASDET.dat` con corte de control por `nroTk`. La diferencia con el problema anterior está dada en que, por cada ítem (o artículo) tendremos que acceder a `ARTICULOS.dat` para buscar los datos complementarios: la descripción (`desc`) y el precio de venta (`precio`).

El programa es casi idéntico al anterior. Solo que, al principio, abriremos el archivo `ARTICULOS.dat` y al final lo cerraremos. Veamos entonces el programa principal:

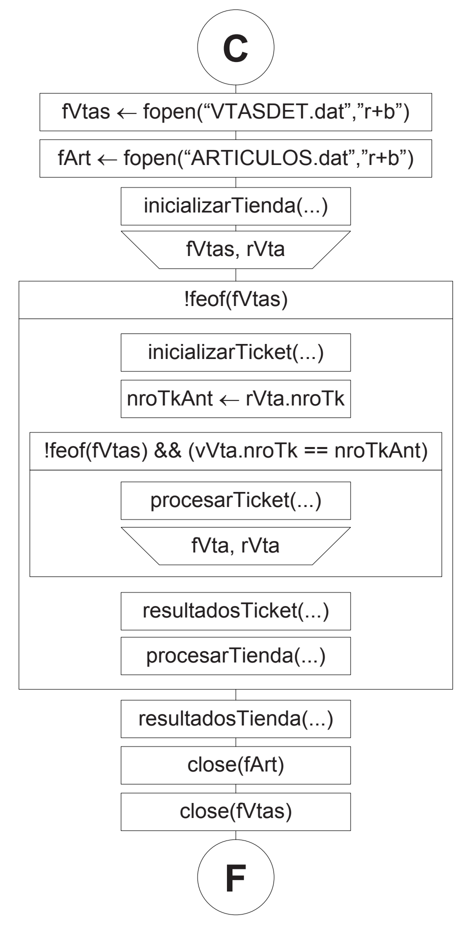


Fig. 10.5 Programa principal.

Como accedemos a dos archivos diferentes decidí cambiar el nombre de las variables `arch` y `reg` de la siguiente manera:

`fVta`	Archivo de ventas	antes era: `arch`
`fArt`	Archivo de artículos	
`rVta`	Registro del archivo de ventas	antes era: `reg`
`rArt`	Registro del archivo de artículos	

Veamos los módulos `inicializarTicket`, `procesarTicket` y `resultadosTicket`.

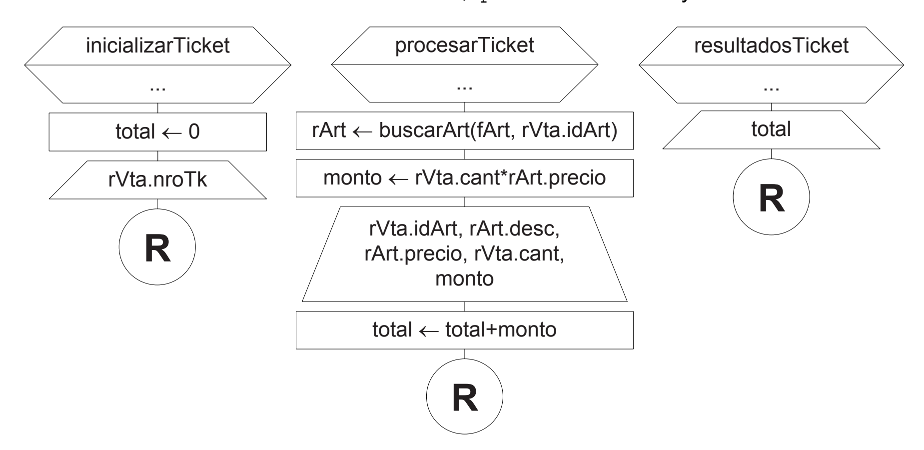


Fig. 10.6 Módulos a nivel grupal.

Básicamente, estos tres módulos resuelven la parte del listado que se emite por cada uno de los *tickets* del archivo. En `inicializarTicket` (es decir, antes de comenzar a procesar los registros de cada *ticket*), mostramos el número de *ticket*.

En `procesarTicket` mostramos cada uno de los artículos que componen la venta y para esto, necesitamos obtener algunos datos complementarios: la descripción y el precio. Por esto, comenzamos el módulo invocando a `buscarArt` que, en función de `rVta.idArt` realizará una búsqueda binaria sobre el archivo `fArt` y retornará un registro con toda la información referente al artículo buscado y encontrado.

Aceptaremos que no existen problemas de inconsistencia de datos, por lo tanto, no habrá registros en `VTASDET.dat` cuyo `idArt` no se encuentre en `ARTICULOS.dat`. Por esto, la función `buscarArt` no requiere de un argumento por referencia para notificarnos si existe o no el artículo buscado. Siempre existirá.

Veamos ahora los módulos que resuelven la segunda y última parte del listado:

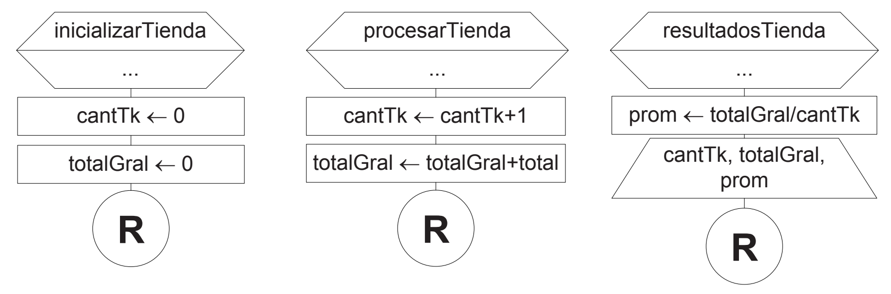


Fig. 10.7 Módulos a nivel general.

Antes de continuar, le recomiendo al lector detenerse a analizar la salida de este programa. ¿En qué orden considera que se emitirán los ítems del listado que se emite para cada uno de los *tickets*?

10.2.2 Uso de arrays auxiliares

En el problema anterior, emitimos un listado detallando, por cada *ticket*, los artículos que lo componen. El orden en el que aparecerán los ítems coincidirá con el orden que mantengan los registros en el archivo `VTASDET.dat`. Es decir que si en el archivo los registros se encuentran ordenados por `nroTk` y luego (por cada *ticket*), ordenados por `idArt`, entonces los ítems en el listado aparecerán ordenados por código de artículo. Sin embargo, como el enunciado no aclara nada sobre un segundo nivel de ordenamiento de los registros del archivo, debemos suponer que el listado no respetará ningún orden en particular.

Para que los ítems del listado se impriman en un orden diferente al que mantienen los registros del archivo tendremos que almacenarlos en memoria (en un *array*), ordenarlos y, luego, emitirlos.

Problema 10.3

Ídem al problema anterior, pero el listado de los artículos de cada *ticket* debe salir ordenado por descripción.

Nota: se considera que un *ticket* no tendrá más de 50 artículos diferentes (ítems).

Por ejemplo:

Número de Ticket: 9999				
Id. Artículo	*Descripción*	*Precio de Venta*	*Cantidad*	*Monto*
9999	Aceite de oliva...	99.99	99	99.99
9999	Máq. caminadora...	99.99	99	99.99
9999	Notebook 2GB...	99.99	99	99.99
:	:	:	:	:
			Total Ticket:	999.99
Cantidad total de tickets procesados:		99		
Monto total facturado		99.99		
Monto promedio facturado		99.99		

Análisis

Como analizamos más arriba, en este caso no podremos listar directamente los artículos de cada *ticket* como lo hicimos en `procesarTicket` en el problema anterior.

Por cada artículo (o ítem), luego de buscar su registro en `ARTICULOS.dat`, lo agregaremos a un *array* con capacidad para mantener 50 registros de tipo `RArr`.

```
typedef struct RArr
{
   int idArt;
   int cant;
   double precio;
   char desc[50];
}RArr;
```

La estrategia será recorrer `VTASDET.dat` con corte de control por `nroTk`. Por cada *ticket* (grupo de artículos), agregaremos al *array* `arr` cada unos de sus artículos. Luego, en `resultadosTicket` podremos recorrer el *array* para emitir el listado.

Veamos la resolución:

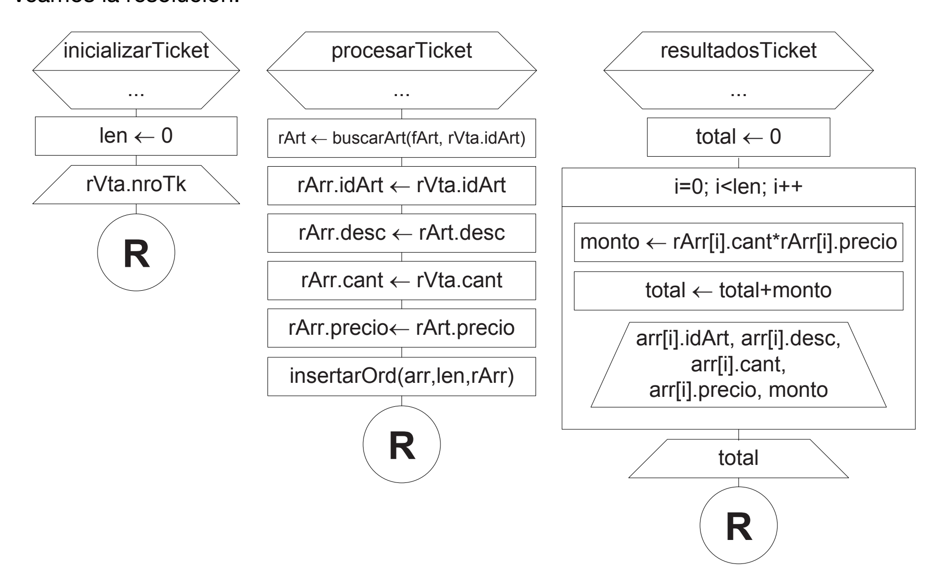


Fig. 10.8 Desarrollo de las funciones del problema 10.3.

En `procesarTicket` buscamos los datos del artículo `rVta.idArt`, asignamos valores a todos los campos de `rArr` (registro de tipo `RArr`) e invocamos a `insertarOrd`. Esta función insertará, en orden alfabético por el campo `desc`, el registro `rArr` en el *array* `arr`.

El programa principal y los módulos `inicializarTienda`, `procesarTienda` y `resultadosTienda` son idénticos a los que desarrollamos en el problema anterior.

10.2.3 Mantener archivos (pequeños) en memoria

En el Capítulo 8, cuando estudiamos las operaciones de entrada/salida sobre archivos, explicamos que, siempre que se pueda, debemos minimizar la cantidad de accesos a los registros de un archivo porque el abuso de este tipo de operaciones trae aparejado un alto costo en materia de rendimiento (*performance*) de la aplicación.

En los problemas anteriores, optamos por realizar búsquedas binarias para acceder al archivo `ARTICULOS.dat` y no consideramos la posibilidad de subir todo su contenido a memoria por dos motivos fundamentales:

1. No conocemos cuál es la cantidad máxima de artículos que la tienda comercializa.
2. Aunque pudiéramos conocer la cantidad anterior, este valor potencialmente podría ser muy grande ya que, tal vez, la tienda comercialice una gran cantidad de artículos de tipos o rubros muy diversos.

Según su naturaleza, cierto tipo de archivos de consulta tendrán una cantidad de registros acotada y, seguramente, pequeña. Esto nos dará la posibilidad de manejar su contenido en memoria, sobre *arrays*.

Problema 10.4

En el mismo contexto de los problemas anteriores, se dispone de los siguientes archivos:

VTASDET.dat	ARTICULOS.dat	RUBROS.dat
typedef struct VtaDet { **int** nroTk; **int** idArt; **int** cant; **int** cantEntr; }VtaDet;	**typedef struct** Articulo { **int** idArt; **char** desc[50]; **double** precio; **int** idRubro; }Articulo;	**typedef struct** Rubro { **int** idRubro; **char** desc[50]; **double** promo; }Rubro;

Los archivos `VTASDET.dat` (ordenado por `nroTk`) y `ARTICULOS.dat` (ordenado por `idArt`) ya son conocidos. Ahora se agrega el archivo `RUBROS.dat`, sin orden y con no más de 20 registros.

Cada artículo registrado en `ARTICULOS.dat` pertenece a un rubro y la relación está dada por el campo `idRubro`.

ARTICULOS.dat

idArt	**desc**	**idRubro**	...
10	Notebook	3	
20	Perfume	5	
30	Mouse	3	
40	Riel Pesca	4	
:	:	:	

RUBROS.dat

idRubro	**desc**	...
:		
3	Computación	
4	Caza y Pesca	
5	Perfumería	
:	:	

En este ejemplo, vemos que los artículos “Notebook” y “Mouse” pertenecen al rubro 3 (computación). El artículo “Riel Pesca” pertenece al rubro “Caza y Pesca” y el artículo “Perfume” pertenece al rubro “Perfumería”.

Es decir, los rubros representan grupos de artículos que, según sus características funcionales, se considera que son elementos de un mismo tipo. Otros ejemplos podrían ser los rubros: “Fotografía”, “Audio y Video”, “Camping”, “Indumentaria”, etcétera.

Cuando la tienda decide promocionar los artículos de un determinado rubro utiliza el campo `promo`. Si un rubro está en promoción entonces este campo tendrá un valor mayor que 0 y menor que 1. Por ejemplo: `promo =` 0.75 significa que todos los artículos del rubro tendrán un 25% de descuento sobre el precio indicado en `ARTICULOS.dat`. En cambio, el valor `promo` para aquellos rubros que no están en promoción siempre será 1. Es decir que el precio final de un determinado artículo se puede calcular multiplicando su precio de venta por el valor `promo` del rubro al cual pertenece.

Se pide:

1. Por cada *ticket* emitir el siguiente listado:

Número de Ticket: 9999

Artículo	*Descripción*	*Precio*	*Descuento*	*Cant.*	*Monto*
9999	Aceite de oliva...	99.99	99.99	99	99.99
9999	Máq. caminadora...	99.99	99.99	99	99.99
9999	Notebook 2GB...	99.99	99.99	99	99.99
:	:	:	:	:	:
				Total Ticket:	999.99

Monto Bruto (sin descuentos):	999.99
Total Descuentos:	999.99
Importe Neto facturado:	999.99

Rubro	*Descuento*
XXXXXXX	99.99
XXXXXXX	99.99
:	:

2. Emitir, al finalizar el proceso, un listado discriminando por cada rubro los montos totales descontados.

Rubro	*Descuento*	*Representa %*
XXXXXXX	99.99	99
XXXXXXX	99.99	99
:	:	:
Total:	999.9	100

Análisis

El problema es prácticamente igual al problema anterior. Solo que para imprimir los artículos de cada *ticket* necesitaremos conocer, por cada uno, el factor `promo` que, multiplicado por su precio, nos dará el precio de venta final. Esta información la encontraremos en el archivo `RUBROS.dat`.

Dado que `RUBROS.dat` tiene, a lo sumo, 20 registros, vamos a optar por “subir” el archivo a un *array*, cuya estructura veremos a continuación. Esto nos permitirá agilizar el acceso a su contenido.

```
typedef struct RArrRubro
{
   int idRubro;         // codigo de rubro
   char desc[50];       // descripcion del rubro
   double promo;        // promocion
   double acumDtoTk;    // acumulador de descuentos por ticket
   double acumDtoGral;  // acumulador de descuentos en general
}RArrRubro;
```

Esta estructura nos permite disponer de la información contenida en el archivo de rubros y, además, mantener dos acumuladores por cada rubro: `acumDtoTk` lo utilizaremos para acumular los descuentos aplicados a los artículos de cada *ticket*. Su valor debe inicializarse `inicializarTicket`. En cambio, `acumDtoGral` lo utilizaremos para acumular los descuentos otorgados a todos los *tickets* de `VTASDET.dat`.

El programa principal es casi idéntico al de los ejercicios anteriores ya que se trata de un problema de corte de control sobre `VTASDET.dat`. Solo agregaremos, al principio, la invocación a la función `subirRubros` que sube a un *array* todo el contenido de `RUBROS.dat`.

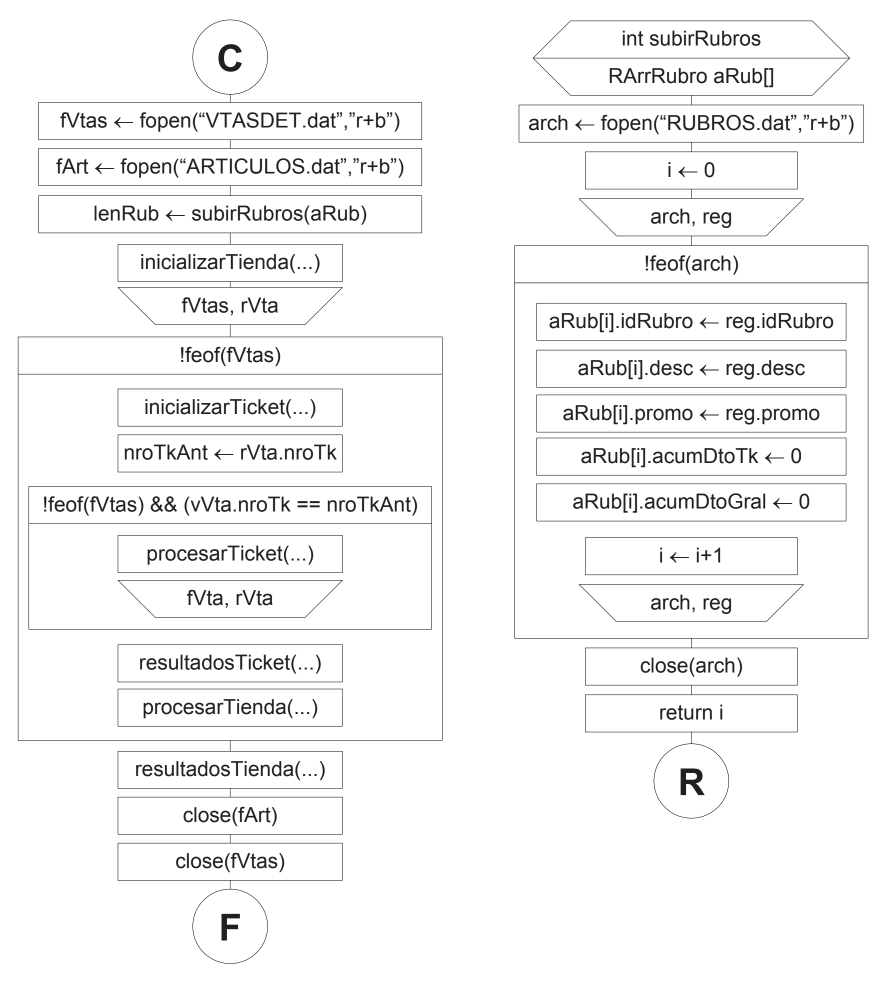


Fig. 10.9 Programa principal, sube archivo a memoria.

La función `subirRubros` retorna la longitud del *array* `aRub`, que coincidirá con la cantidad de rubros definidos en el archivo.

Vamos a modificar la estructura del *array* `arr`, que usamos para almacenar temporalmente los ítems de cada *ticket*, de forma tal que también nos permita mantener en memoria el descuento aplicado a cada ítem.

```
typedef struct RArr
{
   int idArt;
   int cant;
   double precio;
   char desc[50];
   double dto;    // agrego el descuento
}RArr;
```

Veamos ahora los módulos `inicializarTicket` y `procesarTicket` en donde obtenemos el descuento que corresponde aplicar a cada artículo y acumulamos en los campos `acumDtoTk` y `acumDtoGral` del *array* de rubros `aRub`.

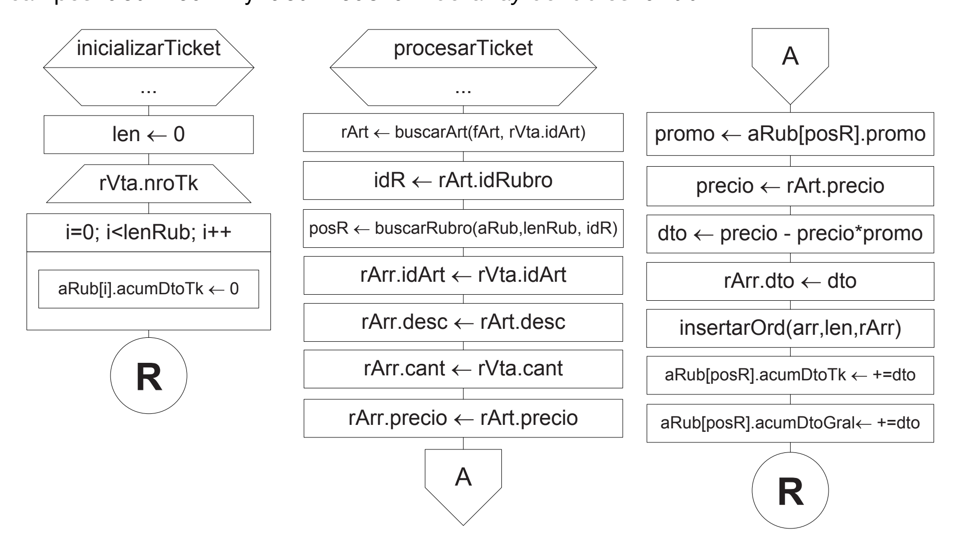


Fig. 10.10 Módulo `procesarTicket`.

Notemos que en `inicializarTicket` solo inicializamos el campo `acumDtoTk`. El campo `acumDtoGral` lo inicializaremos en `inicializarTienda`.

Veamos ahora el módulo `resultadosTicket` donde emitimos el listado del *ticket* que acabamos de procesar.

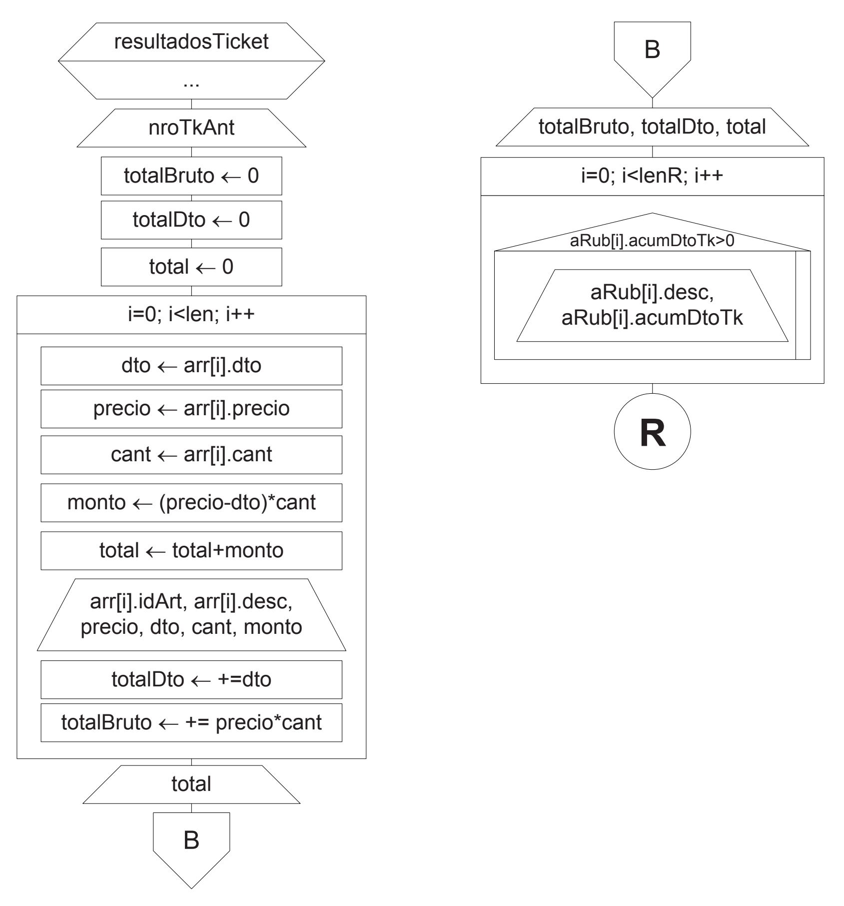


Fig. 10.11 Desarrollo de la función `resultadosTicket`.

Ahora tenemos que desarrollar los módulos `inicializarTienda`, `procesarTienda` y `resultadosTienda` para emitir el listado de los descuentos otorgados según el rubro.

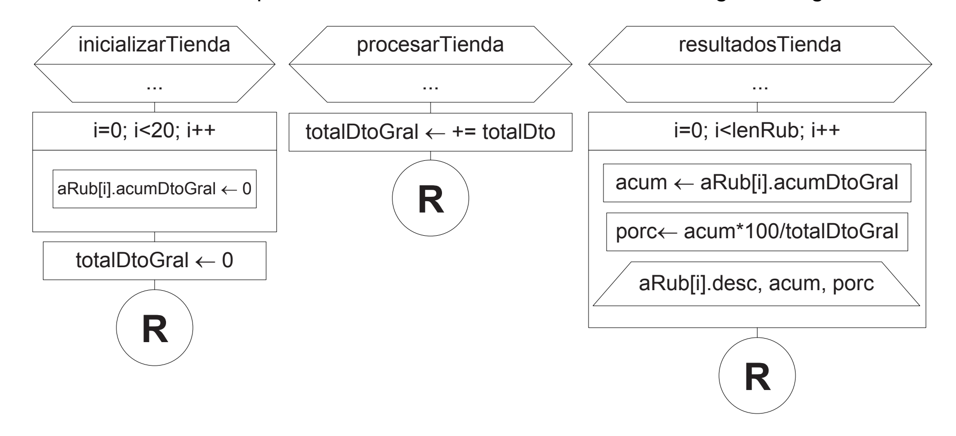


Fig. 10.12 Desarrollo de las funciones del problema 10.4.

10.3 Apareo de archivos

Cuando tenemos dos o más archivos, todos ordenados por el mismo campo, podemos recorrerlos simultáneamente para intercalar sus registros o bien para determinar cuáles registros aparecen en un archivo y no en los otros.

Supongamos que tenemos dos archivos, `A.dat` y `B.dat` donde cada uno contiene registros de tipo `int`. Ambos archivos se encuentran ordenados ascendentemente.

A.dat	B.dat
1	2
3	3
4	6
eof	7
	8
	eof

Como vemos, los dos archivos están ordenados en forma ascendente. Leemos el valor del primer registro de `A` (lo llamaremos `a`) y luego el valor del primer registro de `B` (lo llamaremos `b`).

Si `a` (que vale 1) es menor que `b` (que vale 2) podemos determinar que el valor `a` está en el archivo `A`, pero no en el archivo `B` ya que, como ambos archivos están ordenados, los valores subsiguientes de `B` serán, sin dudas, mayores que `a`. Luego leemos el próximo registro del archivo de `A` y lo comparamos con `b`.

Si `a` (que ahora vale 3) es mayor que `b` (que vale 2) podemos determinar que `b` está en el archivo `B`, pero no en el archivo `A`. Luego leemos el siguiente registro de `B`.

Si `a` (que vale 3) es igual a `b` (que ahora vale 3) entonces diremos que el valor se encuentra en ambos archivos. En este caso, leemos `A` y `B` y continuamos con el proceso hasta que finalice alguno de los dos archivos.

Si un archivo termina antes que el otro podemos asegurar que todos los registros sobrantes que quedan en el archivo que aún no finalizó no están contenidos en el archivo que terminamos de leer primero.

El algoritmo de apareo de archivos consiste en leer el primer registro de cada archivo, comparar su valor para determinar cuál es el menor, procesarlo y luego avanzar al siguiente registro.

Notemos que es fundamental el hecho de que ambos archivos se encuentren ordenados. De lo contrario, no sería posible aplicar el algoritmo.

El siguiente esquema permite aparear dos archivos.

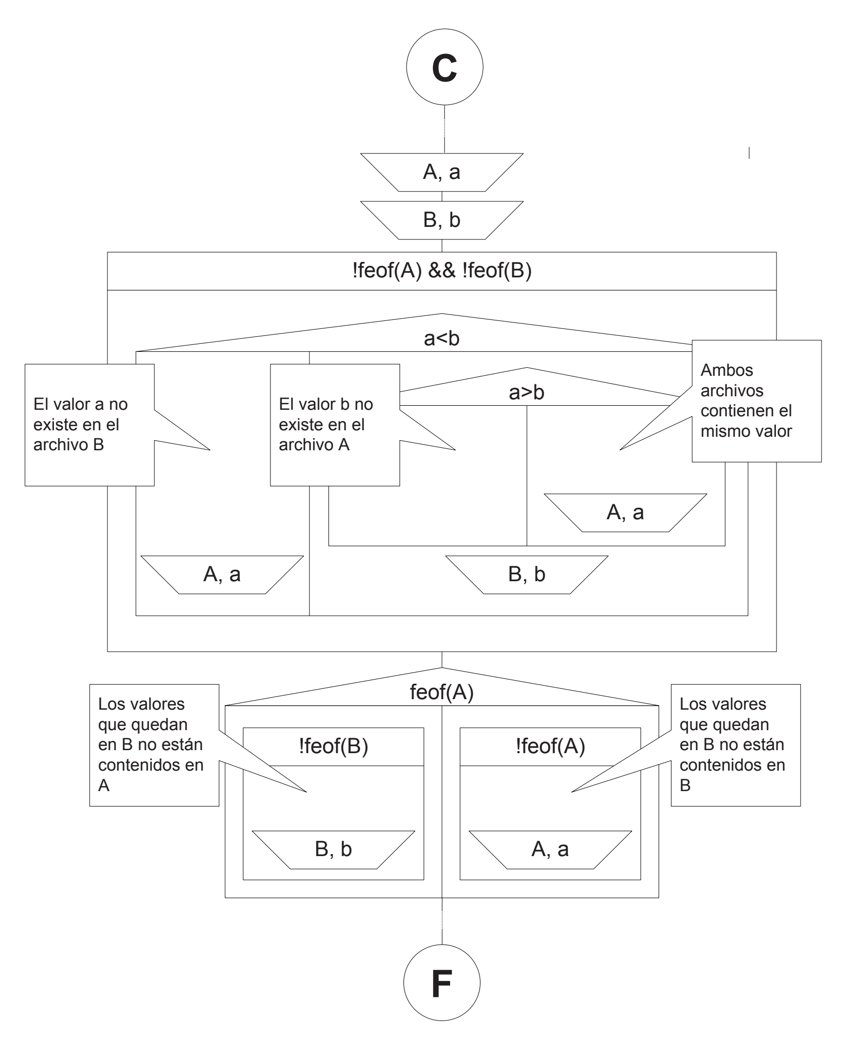


Fig. 10.13 Esquema de un apareo de 2 archivos.

Siguiendo el esquema anterior desarrollaremos un programa que, a partir de los archivos `A` y `B`, genere un nuevo archivo `C` intercalando los registros de los otros 2 (sin repetición) y un cuarto archivo `D` conteniendo los valores comunes a ambos archivos.

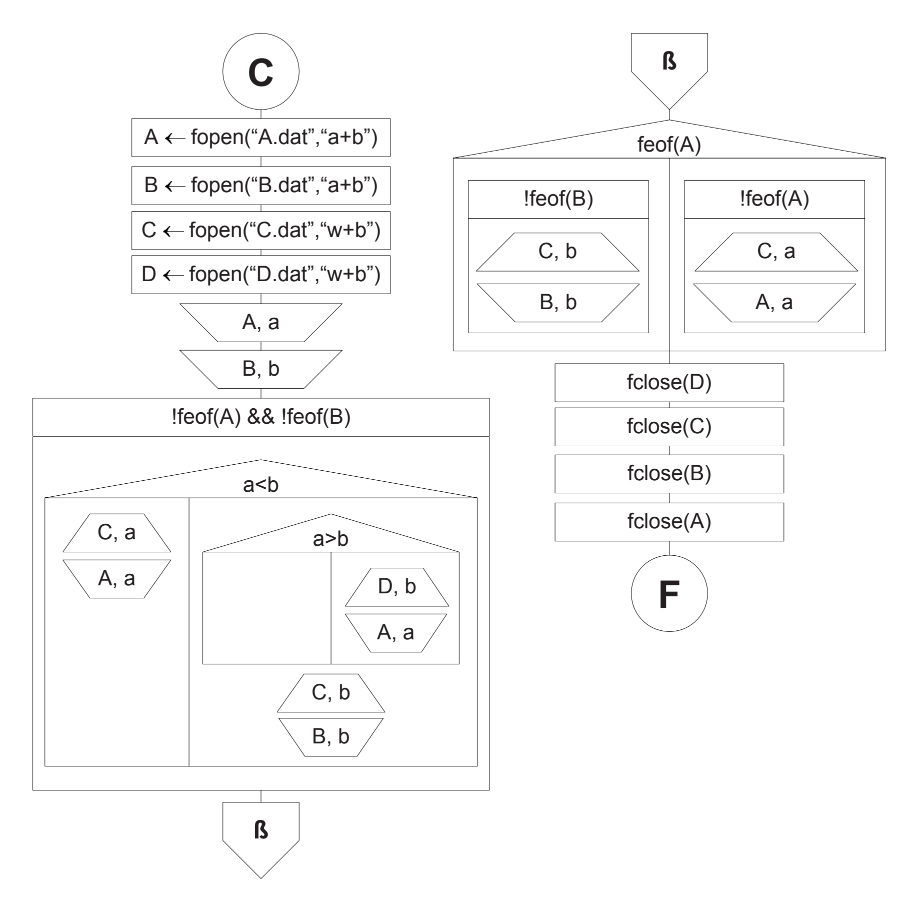
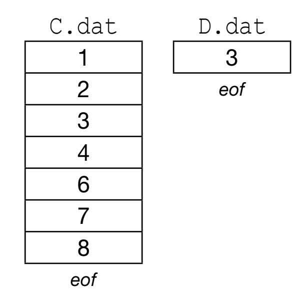


Fig. 10.14 Apareo de archivos.

La salida del algoritmo será:



Problema 10.5

En el mismo contexto de los problemas anteriores, la tienda requiere el desarrollo de un programa que le permita actualizar los precios de venta de sus artículos. Para esto, cuenta con los archivos `ARTICULOS.dat` (ya conocido) y `PRECIOS2011.dat` cuya estructura veremos a continuación. Los archivos están ordenados por el campo `idArt`.

ARTICULOS.dat	PRECIOS2011.dat
`typedef struct Articulo { int idArt; char desc[50]; double precio; int idRubro; }Articulo;`	`typedef struct Precio { int idArt; double precio; }Precio;`

```
typedef struct Articulo
{
   int idArt;
   char desc[50];
   double precio;
   int idRubro;
}Articulo;
```

```
typedef struct Precio
{
   int idArt;
   double precio;
}Precio;
```

Como ya sabemos, el archivo de artículos tiene un registro por cada uno de los artículos que la tienda comercializa. En cambio, el archivo de precios tendrá un registro por cada uno de los artículos cuyo precio de venta debe ser modificado. Es decir que, muy probablemente, existan registros en `ARTICULOS.dat` que no tengan un registro relacionado en `PRECIOS2011.dat`.

Vamos a suponer también que en el archivo de precios puede haber registros cuyo `idArt` no tenga relación con ninguno de los registros del archivo de artículos. Estos registros se considerarán erróneos.

Se pide:

1. Desarrollar un programa que actualice los precios de venta de los artículos de la tienda con los nuevos precios detallados en `PRECIOS2011.dat`.
2. Para aquellos artículos cuyo precio de venta resulte con un incrementó del 20% o más, emitir un listado con el diseño que se detalla a continuación.

Código	*Descripción*	*Incremento %*
999	XXXXXX	99
999	XXXXXX	99
:	:	:

3. Generar el archivo `ERRORES2011.dat` con los registros erróneos que contenga el archivo de precios.

Análisis

Comencemos viendo un ejemplo de los datos que podrían contener los archivos.

ARTICULOS.dat

idArt	**desc**	**precio**	**...**
10	Notebook	700	
20	Perfume	250	
30	Mouse	20	
40	Riel Pezca	60	
50	Filmadora	500	
60	LCD 42”	2500	
70	Bicicleta	350	
:	:	:	

PRECIOS2011.dat

idArt	**precio**	**...**
20	750	
30	15	
35	1500	
40	150	
65	2350	
eof		

En principio, podemos observar que, en `PRECIOS2011.dat`, los registros cuyo `idArt` es 35 y 65 no tienen ningún artículo relacionado en `ARTICULOS.dat`. Es decir, se trata de registros erróneos que debemos grabar en `ERRORES2011.dat`.

También vemos que los únicos artículos cuyos precios serán modificados son los artículos con `idArt`: 20, 30 y 40.

El problema se resuelve recorriendo ambos archivos a la vez, comparando la relación que existe entre los valores de sus campos `idArt`.

Llamemos `rArt` a la variable que utilizaremos para leer el archivo `ARTICULOS.dat` y `rPre` a la variable que utilizaremos para leer `PRECIOS2011.dat`. Entonces:

- Si `rArt.idArt` es menor que `rPre.idArt`, significa que el artículo que leímos desde el archivo de artículos no existe en el archivo de precios. Es decir: el precio del artículo no será modificado.
- Si `rArt.idArt` es mayor que `rPre.idArt`, significa que el artículo que leímos desde el archivo de precios no existe en el archivo de artículos. Entonces se trata de un registro erróneo que debe ser grabado en `ERRORES2011.dat`.
- Si `rArt.idArt` es igual que `rPre.idArt`, significa que tenemos que actualizar el precio del artículo representado en `rArt` con el valor indicado en `rPre`. Además, si el nuevo valor representa una subida del 20% o más, el artículo deberá aparecer en el listado.

Siguiendo el mismo esquema de apareo de archivo estudiado más arriba, llegaremos al siguiente algoritmo de solución:

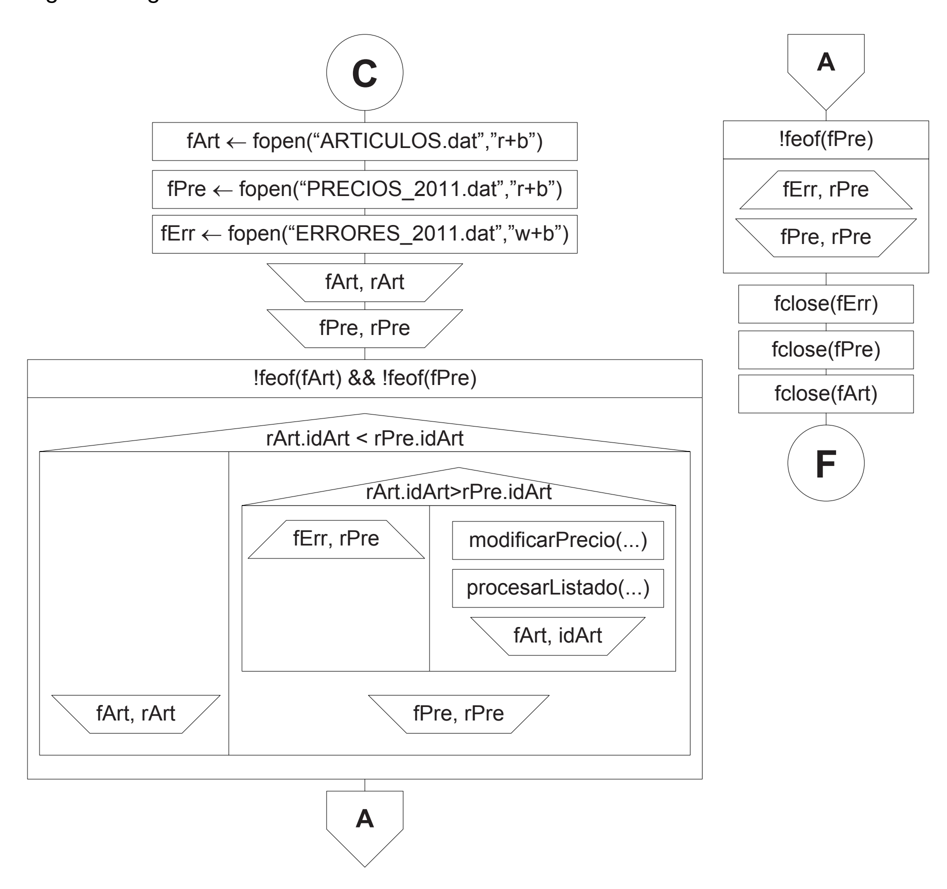


Fig. 10.15 Solución al problema 10.5.

Notemos que al llegar el *eof* de alguno de los dos archivos, solo nos interesará procesar los registros que pudieran haber quedado pendientes en `PRECIOS2011.dat` ya que todos estos estarán representando modificaciones de precios de artículos inexistentes.

Veamos ahora los módulos `modificarPrecio` y `procesarListado`:

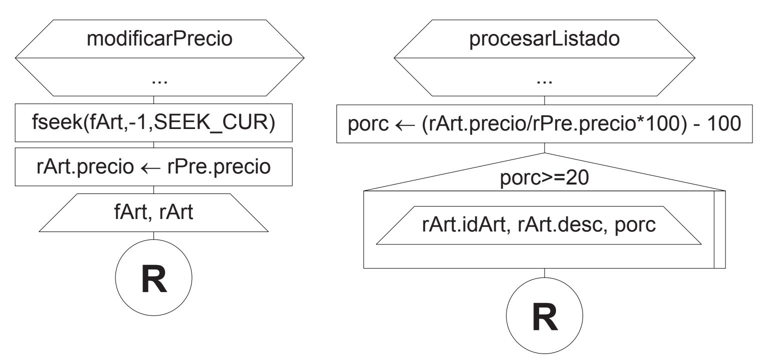


Fig. 10.16 Solución al problema 10.5 (continuación).

Problema 10.6

Ídem al anterior, pero el listado debe salir ordenado decrecientemente según el porcentaje de aumento. Se considera que no habrá más de 100 artículos cuyos precios se incrementarán por encima del 20%.

El análisis y posterior desarrollo de este ejercicio quedará a cargo del lector.

10.3.1 Apareo de archivos con corte de control

Problema 10.7

En el mismo contexto de todos los problemas anteriores, se cuenta con los siguientes archivos:

VTASCAB.dat	VTASDET.dat	ARTICULOS.dat
typedef struct VtaCab { **int** nroTk; **long** fecha; **int** idCli; **int** importe; }VtaCab;	**typedef struct** VtaDet { **int** nroTk; **int** idArt; **int** cant; **int** cantEntr; }VtaDet;	**typedef struct** Articulo { **int** idArt; **char** desc[50]; **double** precio; **int** idRubro; }Articulo;

`VTASCAB.dat` y `VTASDET.dat` se encuentran ordenados por `nroTk`.`ARTICULOS.dat` se encuentra ordenado por `idArt`.

Nota: Se considera que pueden existir los siguientes casos erróneos:

1. Registros en `VTASCAB.dat` sin su correspondiente detalle en `VTASDET.dat`.
2. Registros en `VTASDET.dat` sin su correspondiente cabecera en `VTASCAB.dat`.

Los registros erróneos de `VTASCAB.dat` deben grabarse en un nuevo archivo llamado `ERRCAB.dat`. Análogamente, los registros con error de `VTASDET.dat` deben grabarse en el archivo `ERRDET.dat`.

Se pide emitir, por cada *ticket*, el siguiente listado:

Nro. Ticket: 999 *Fecha: 9999999* *Cliente: 99999* *Total: $99999*

Código	*Descripción*	*Precio*	*Cant.*	*Monto*
999	XXXX	99.99	99	999.99
999	XXXX	99.99	99	999.99
:	:	:	:	:

Total: $99999

Análisis

En este problema, se agrega el archivo `VTASCAB.dat` que contiene un registro por cada uno de los *tickets* que han sido emitidos en la tienda. La relación entre `VTASCAB.dat` (archivo de cabecera) y `VTASDET.dat` (archivo de detalles) la explicaremos a continuación.

Cada venta se compone de una cabecera y de uno o varios detalles. La cabecera es única y contiene la fecha, el importe total y el código de cliente al que se le facturó. En cambio, el detalle se compone de uno o varios ítems y, como ya sabemos, cada ítem describe un artículo y sus cantidades adquiridas y entregadas.

Veamos un ejemplo de estos archivos:

`VTASCAB.dat`

nroTk	**fecha**	**idCli**	**importe**
1	20100530	50	250
2	20100530	20	180
3	20100601	40	630
:	:	:	:

`VTASDET.dat`

nroTk	**idArt**	**cant**	**...**
1	10	1	...
1	40	1	...
1	20	3	...
2	30	1	...
3	20	1	...
3	10	2	...
:	:	:	:

En este ejemplo, vemos que el *ticket* número 1 corresponde a una venta que se le efectuó al cliente, cuyo `idCli` es 50, el día 30 de mayo de 2010 por un importe total de $250. Además, vemos en `VTASDET.dat` que esta venta se compone de 1 unidad de los artículos 10 y 40 y 3 unidades del artículo 20.

El problema se resuelve con un apareo entre los archivos `VTASCAB.dat` y `VTASDET.dat` que nos permitirá determinar cuáles registros son erróneos. Para el caso en el que ambos registros leídos tengan el mismo `nroTk`, recorreremos `VTASDET.dat` con corte de control para emitir los ítems del listado solicitado.

Veamos la solución:

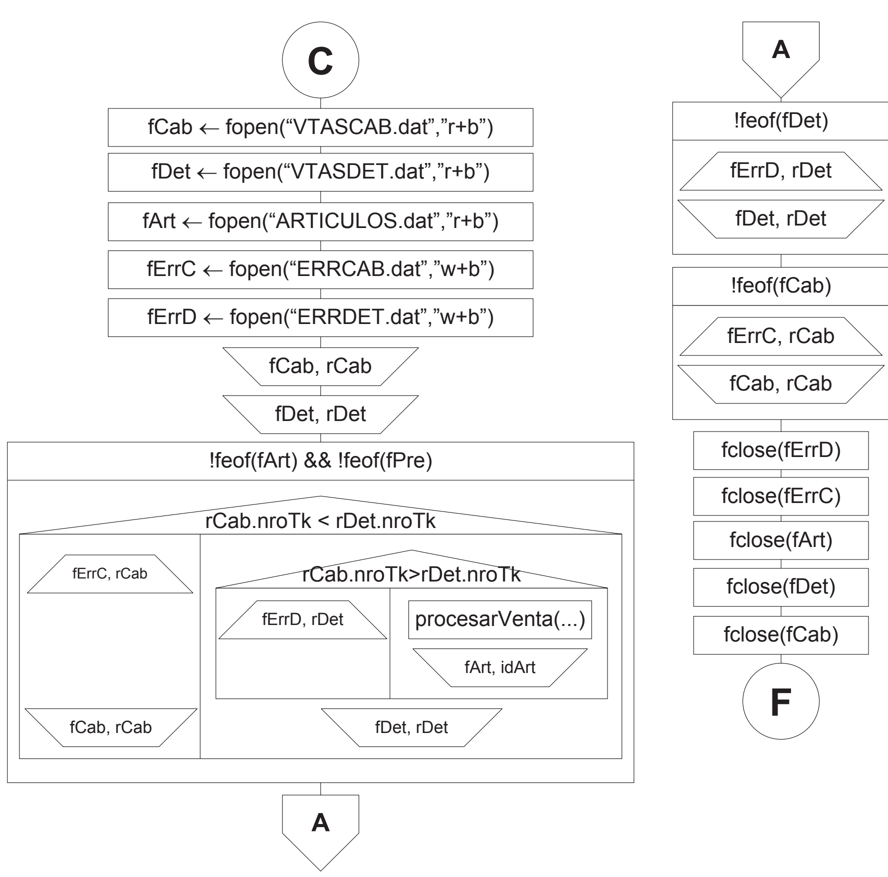


Fig. 10.17 Solución al problema 10.7.

Como explicamos más arriba, si `rCab.nroTk` es igual a `rDet.nroTk` entonces tenemos una venta correctamente registrada y para emitir el listado recorreremos `VTASDET.dat` con corte de control. Esto lo hacemos en `procesarVenta`.

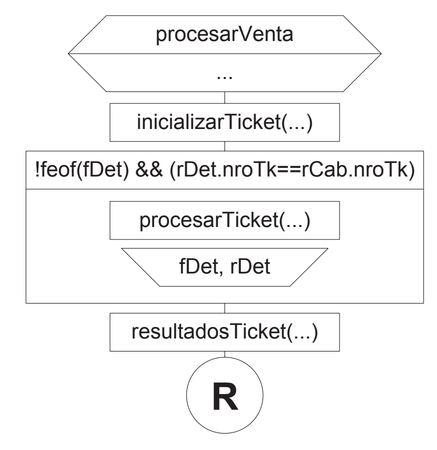


Fig. 10.18 Solución al problema 10.7 (continuación).

El desarrollo de los módulos `inicializarTicket`, `procesarTicket` y `resultadosTicket` queda a cargo del lector.

Problema 10.8

Ídem al anterior, pero considerando que puede haber ventas con artículos inexistentes. En este caso, la venta debe considerarse errónea y no debe aparecer en el listado. Considere que, a lo sumo una venta tiene 20 ítems.
Por cada venta que contenga este tipo de error se debe generar un registro en el archivo `VTASERR.dat` con la siguiente estructura de registro.

```
typedef struct VtaErr
{
   int nroTk;
   int idArt;
}VtaErr;
```

Análisis

El hecho de que exista la posibilidad de que un registro de `VTASDET.dat` haga referencia a artículo inexistente en `ARTICULOS.dat` nos obliga a almacenar en memoria todos los ítems del listado para verificar, antes de imprimirlo, que ninguno sea erróneo.
Para resolver este problema, solo debemos modificar el módulo `procesarVenta` del diagrama anterior y, obviamente, de abrir y cerrar el archivo `VTASERR.dat` en el programa principal. A este archivo, lo llamaremos `fErrV`.
Recordemos el diagrama de `procesarVenta`:

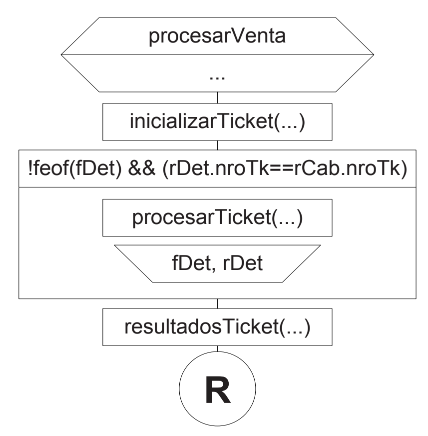


Fig. 10.19 Solución al problema 10.8

La estrategia será la siguiente: en `procesarTicket` buscaremos sobre `ARTICULOS.dat` un artículo cuyo código coincida con `rDet.idArt`. Si lo encontramos agregaremos en un *array* un registro con los datos del ítem leído. El *array* tendrá 20 elementos y la siguiente estructura de registro:

```
typedef struct RArr
{
   int idArt;
   int cant;
   char desc[50];
   double precio;
}Rarr;
```

En cambio, si no encontramos ningún artículo cuyo código coincida con `rDet.idArt` grabaremos en `fErrV` un registro indicando el número de *ticket* y el código de artículo erróneo. En este caso, el listado del *ticket* ya no se deberá emitir así que utilizaremos una variable *booleana* (`hayError`) que nos permitirá determinar, en `resultadosTicket`, si imprimir o no el contenido del *array*.

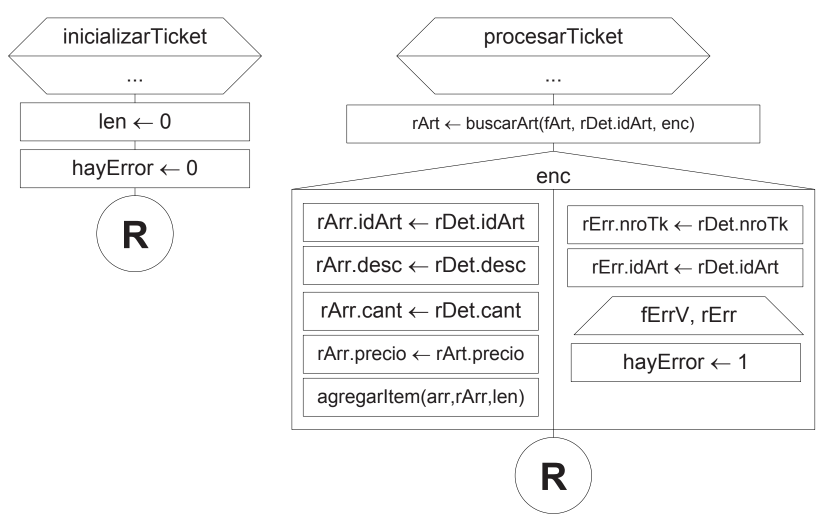


Fig. 10.20 Solución del problema 10.8 (continuación).

Veamos ahora el desarrollo de `resultadosTicket`:

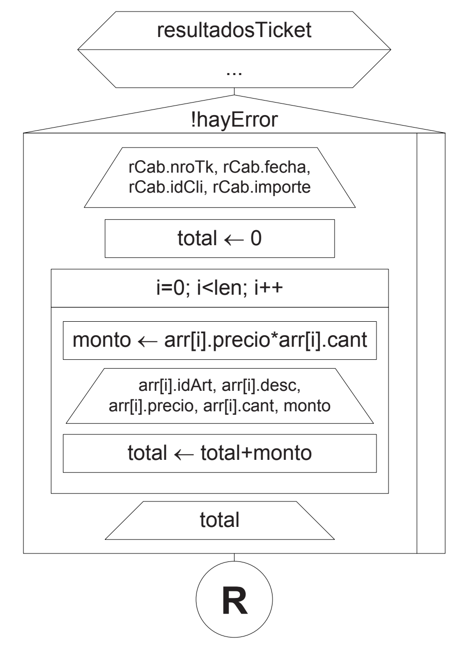


Fig. 10.21 Solución del problema 10.8 (continuación).

10.4 Resumen

Con este capítulo cerramos la primera parte del libro analizando y resolviendo problemas cuya solución requirió aplicar todos los conceptos y conocimientos adquiridos desde el Capítulo 1 hasta el presente.

Queda pendiente aún el estudio de estructuras dinámicas lineales, tema que veremos a continuación para pasar, luego, a la segunda parte de este trabajo cuyo objetivo será el estudio de estructuras y algoritmos con mayor nivel de complejidad.

10.5 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

10.5.1 Mapa conceptual

10.5.2 Autoevaluaciones

10.5.3 Presentaciones*

11

Estructuras de datos dinámicas lineales

Contenido

Objetivos del capítulo

- Comprender la diferencia entre estructuras estáticas y dinámicas, lineales y no lineales.
- Entender cómo pasar punteros por referencia a una función.
- Analizar la estructura “lista enlazada” y sus operaciones asociadas: insertar, buscar, eliminar, etcétera.
- Estudiar estructuras restrictivas: Pila (LIFO) y Cola (FIFO), y sus operaciones asociadas.
- Estudiar otras estructuras dinámicas lineales: lista doblemente enlazada, lista circular, etcétera.
- Combinar estructuras diferentes estructuras de datos: *array* de listas, lista con sublistas, matriz de pilas, etcétera.

Competencias específicas

- Conocer, identificar y aplicar las estructuras dinámicas lineales en la solución de problemas del mundo real.

11.1 Introducción

Básicamente, cuando hablamos de "estructura" nos referimos a "colección". De hecho, la palabra `struct` de C nos permite definir un tipo de datos nuevo con base en una colección de otros tipos de datos ya existentes. Por ejemplo, veamos el siguiente fragmento de código:

```
typedef struct Persona
{
   int dni;
   char nombre[20];
   long fechaNac;
}Persona;
```

Aquí definimos el tipo de datos `Persona` (o `struct Persona`) como una colección de tres tipos de datos primitivos:

```
Persona = {int, char[20], long}
```

Vimos también que los *array* son un caso de tipo de dato estructurado que nos permite mantener en memoria una colección finita y acotada de valores del mismo tipo. Por ejemplo:

```
int aEnteros[3];
aEnteros[0] = 10;
aEnteros[1] = 20;
aEnteros[2] = 30;
```

En este código, al definir la variable `aEnteros` como un `int[3]` podemos almacenar en memoria hasta 3 valores enteros. Gráficamente, lo representamos de la siguiente manera:

0	10
1	20
2	30

Fig. 11.1 Representación gráfica de un `int[3]`.

En el siguiente ejemplo declaramos un *array* con capacidad para contener hasta 100 "personas" aunque, por el momento, solo asignamos un único valor (una sola persona).

```
// declaramos el array
Persona aPersonas[100];

// asignamos un valor (una fila)
aPersona[0].dni = 23112342;
strcpy(aPersona[0].nombre,"Juan");
aPersona[0].fechaNac = 20110325;
```

Lo representamos de la siguiente manera:

	dni	nombre	fechaNac
0	23112342	Juan	20110325
1			
:			
99			

Fig. 11.2 Representación gráfica de una `persona[100]`.

En todos estos ejemplos, hablamos de colecciones ya que, como comentamos más arriba, el concepto de "estructura" hace referencia a "colección".

11.2 Estructuras estáticas

Hasta aquí trabajamos con estructuras de datos estáticas, es decir, colecciones cuya capacidad máxima debe definirse previamente. Por ejemplo, al declarar una variable de tipo `Persona[100]` (un *array* con capacidad para contener hasta 100 personas) nuestro programa reservará una cantidad de memoria fija más allá de que durante su ejecución la utilice o no. Repasemos la definición del tipo `Persona`:

```
typedef struct Persona
{
   int dni;         // 2 bytes
   char nombre[20]; // 20 bytes
   long fechaNac;   // 4 bytes
}Persona;
```

Cada registro `Persona` utiliza 26 *bytes*. Así, un *array* de 100 personas ocupará 260 *bytes* de memoria.

11.3 Estructuras dinámicas

El concepto de “estructura dinámica” también se refiere a una colección de valores del mismo tipo. La diferencia está dada en que la cantidad de elementos de la colección puede variar durante la ejecución del programa aumentando o disminuyendo y, en consecuencia, utilizando mayor o menor cantidad memoria.

11.3.1 El nodo

Las estructuras dinámicas se forman “enlazando” nodos. Un nodo representa un conjunto de uno o más valores, más un puntero haciendo referencia al siguiente nodo de la colección.

Veamos cómo definir un nodo en C.

```
typedef struct Nodo
{
   int valor;        // valor que contiene el nodo
   struct Nodo* sig; // referencia al siguiente nodo
}Nodo;
```

Como podemos ver, un nodo simplemente es una estructura que define valores más una referencia (puntero de tipo `Nodo*`) para apuntar al siguiente nodo de la colección.

11.4 Listas enlazadas

Con la estructura `Nodo` ya definida y analizada, fácilmente podemos visualizar una lista enlazada de nodos en la que cada nodo contiene un valor y una referencia al siguiente elemento de la colección.

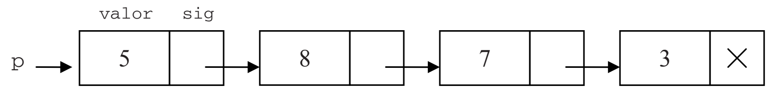


Fig. 11.3 Lista enlazada.

En la figura vemos una lista formada por un nodo con valor 5 cuya referencia apunta a un nodo con valor 8, cuya referencia apunta a un nodo con valor 7, cuya referencia apunta a un nodo con valor 3, cuya referencia tiene un valor nulo por tratarse del último nodo de la lista o último elemento de la colección.

También podemos ver una variable `p` que apunta al primer nodo. Es decir, `p` debe ser una variable de tipo `Nodo*` de forma tal que pueda contener la dirección del primer elemento de la lista. Como el último nodo no tiene "elemento siguiente", el valor de su referencia al siguiente nodo debe ser `NULL`. Para representar este valor en los gráficos, utilizaremos una X (equis grande) o una cruz.

Decimos entonces que `p` apunta al primer nodo de una lista enlazada o, simplemente, `p` representa una lista enlazada sobre la cual, en breve, veremos cómo agregar, buscar, insertar y eliminar elementos.

11.4.1 Estructuras de datos dinámicas lineales

Una lista enlazada formada por una colección de nodos, como la que definimos más arriba, constituye una estructura de datos dinámica lineal. La linealidad de la estructura se refiere al hecho de que cada elemento de la colección tiene un único elemento anterior y un único elemento posterior salvo, obviamente, los casos particulares del primer y último elemento.

Más adelante, estudiaremos otras estructuras dinámicas lineales como por ejemplo, la Pila (o *stack*) y la Cola (o *queue*).

11.4.2 Estructuras de datos dinámicas no lineales

Si bien este tema corresponde a otro capítulo, al menos debemos mencionar la existencia de estructuras de datos no lineales como por ejemplo, los árboles. El siguiente gráfico representa un árbol binario.

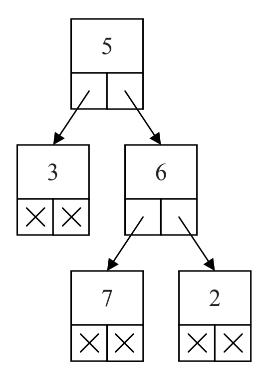


Fig. 11.4 Estructura de datos dinámica no lineal. Árbol binario.

En esta estructura podemos ver que cada nodo (salvo el primero) tiene un único elemento anterior, pero puede tener hasta dos elementos posteriores.

El tema de estructuras dinámicas no lineales lo estudiaremos en detalle más adelante en el capítulo correspondiente.

11.4.3 Punteros por referencia

Antes de comenzar a estudiar las operaciones sobre listas, será conveniente realizar un breve repaso de una de las principales características del lenguaje de programación C: la imposibilidad de que las funciones reciban parámetros por referencia.

En C todos los parámetros que reciben las funciones se pasan por valor salvo, claro, que le pasemos la dirección de memoria en donde dicho valor está alojado.

Por ejemplo, la siguiente función `f` recibe un entero como parámetro e intenta modificar su valor asignándole un 3.

```
void f(int x)
{
   x=3;
}
```

```
int main()
{
   int a=10;
   f(a);
   printf("%d\n",a);
}
```

La asignación `x=3` será efectiva mientras dure la invocación a la función `f`, pero cuando esta finalice el valor original que le hayamos pasado como argumento no resultará modificado ya que la función solo trabaja con una copia de este.

Obviamente, si necesitamos hacer que la función `f` (o cualquier otra función) pueda modificar el contenido de sus parámetros entonces tendremos que trabajar con sus direcciones de memoria.

```
void f(int* x)
{
   *x=3;
}
```

```
int main()
{
   int a=10;
   f(&a);
   printf("%d\n",a);
}
```

Todo esto ya lo hemos estudiado en los capítulos anteriores. Ahora tendremos que analizar qué sucederá si necesitamos que una función modifique el valor de un parámetro de tipo puntero.

Por ejemplo: la función `asignarMemoria` recibe un puntero a entero (`int*`) al que le asigna la dirección de un espacio de memoria recientemente “asignado”.

Incorrecto

```
void asignarMemoria(int* p)
{
   p=(int*)malloc(sizeof(int));
}
```

Correcto

```
void asignarMemoria(int** p)
{
   *p=(int*)malloc(sizeof(int));
}
```

En el primer caso (incorrecto), la función recibe un puntero a entero, esto es: la referencia a un valor entero pero no una copia de su dirección de memoria. Podemos modificar el valor referenciado por `p` pero no podemos modificar su propio valor (la dirección que contiene).

En el segundo caso (correcto), recibimos un “puntero a puntero”. Esto es una referencia a la dirección de un valor entero o, en otras palabras, un puntero por referencia. Luego, dentro de la función, `*p` representa el contenido de `p` que simplemente es la dirección de un valor de tipo `int`.

Más adelante, continuaremos con este tema.

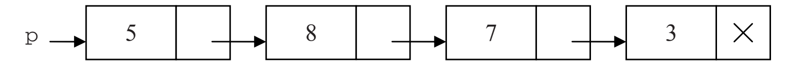
11.5 Operaciones sobre listas enlazadas

Tal como lo hicimos cuando estudiamos *arrays*, definiremos un conjunto de operaciones que nos facilitarán la manipulación de los elementos de las listas.

En todos los casos, trabajaremos sobre listas de enteros cuyos nodos respetan la estructura `Nodo` analizada más arriba.

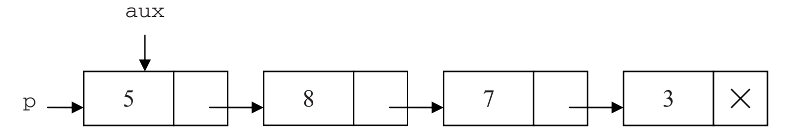
11.5.1 Agregar un elemento nuevo al final de una lista

Analizaremos un algoritmo para agregar un elemento nuevo (nodo) al final de una lista enlazada.



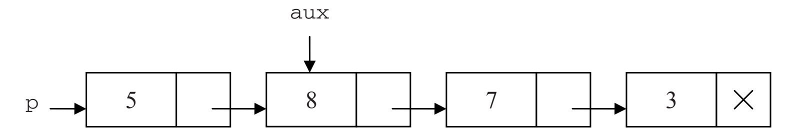
La idea es recorrer la lista avanzando sobre cada uno de sus nodos hasta llegar al último, que es fácilmente identificable por tener el valor nulo `NULL` en su referencia al siguiente nodo.

Para esto, utilizaremos una variable auxiliar `aux` y le asignaremos como valor inicial la dirección contenida por `p`. Luego `aux` apuntará al primer nodo de la lista.



Para saber si `aux` apunta al último nodo de la lista simplemente preguntamos si `aux->sig` es `NULL`. Según el gráfico, `aux->sig` (el siguiente de `aux`) no es `NULL` ya que tiene la dirección del nodo con valor 8 (el segundo elemento).

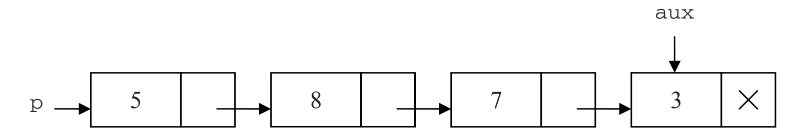
El próximo paso será hacer que `aux` apunte al siguiente nodo. Esto lo logramos asignando a `aux` la dirección de su propio campo `sig` haciendo `aux=aux->sig`.



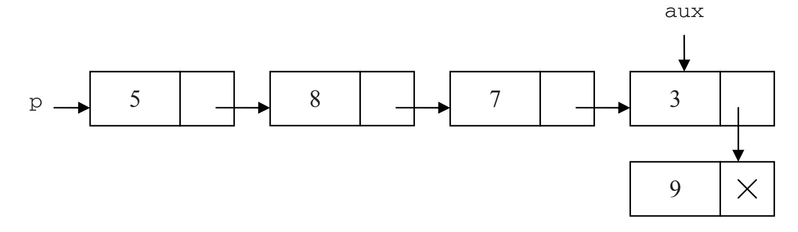
En realidad, este proceso lo haremos dentro de un ciclo de repeticiones que itere mientras que "el siguiente de `aux`" sea distinto de `NULL`.

```
// :
aux=p;
while( aux->sig!=NULL )
{
   aux=aux->sig;
}
// :
```

Como podemos ver, `aux` comienza apuntando al primer nodo de la lista y luego de cada iteración apuntará al siguiente. La condición del `while` se dejará de cumplir cuando `aux` apunte a un nodo sin elemento siguiente que, según nuestro ejemplo, es el nodo con valor 3.



El próximo paso será crear un nuevo nodo y “enlazarlo” al final. Esto es convertirlo en “el siguiente” del último nodo de la lista que, en este momento, está siendo apuntado por `aux`.




```
// creamos un nuevo nodo
Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));

// asignamos su valor y NULL en su siguiente
nuevo->valor=9;
nuevo->sig=NULL;

// lo enlazamos como siguiente de aux
aux->sig=nuevo;
```

Este algoritmo funciona perfectamente si la lista ya tiene al menos un elemento. Analicemos ahora qué sucederá si la lista sobre la que vamos a agregar un nuevo nodo aún está vacía.

p ⟶ ×

Fig. 11.5 Lista vacía.

En la figura vemos que `p` (el puntero al primer nodo de la lista) tiene la dirección nula `NULL`, lo que indica que la lista está vacía. En este caso, luego de crear el nuevo nodo debemos hacer que `p` lo apunte. Para esto, tenemos que asignarle a `p` la dirección de memoria del nuevo nodo ya que, este también será el primero.

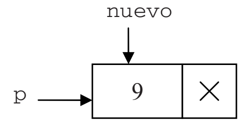


Fig. 11.6 Asignación del primer nodo de la lista.

En principio, esto no debería ser traumático pero, como este algoritmo vamos a encapsularlo dentro de una función, resulta que si `p` es `NULL` tendremos que modificar su valor para asignarle la dirección de memoria del nuevo nodo. Es decir, la función recibirá a `p` por referencia y, dado que `p` es de tipo `Nodo*`, entonces debemos tratarlo como un valor de tipo `Nodo**`.

Veamos el desarrollo de la función `agregar` que recibe por referencia el puntero al primer nodo de la lista más un valor de tipo `int` para agregarlo como último elemento.

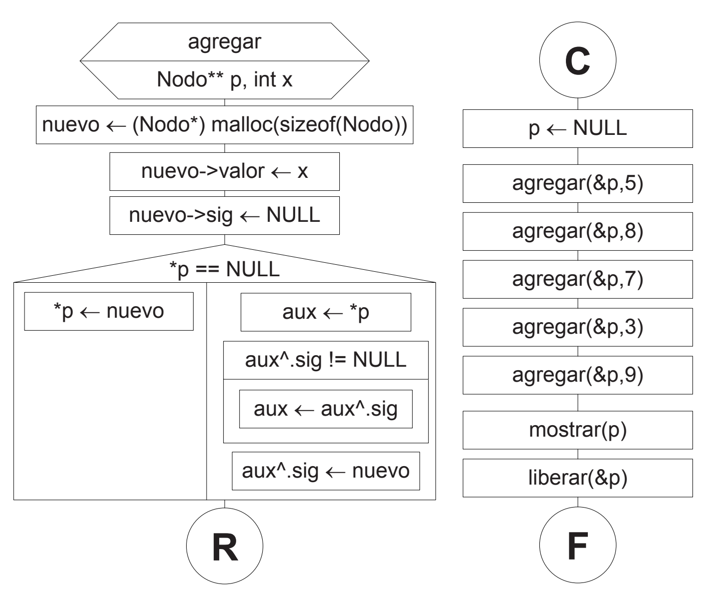


Fig. 11.7 Agrega un elemento al final de una lista enlazada.

Lo primero que hacemos en `agregar` es crear el nuevo nodo y asignarle el valor `x` que recibimos como parámetro y `NULL` como referencia al siguiente. Luego entramos a un `if` para detectar el caso particular en el que la lista está vacía.

Si entramos por la parte derecha del `if` será porque la lista ya tiene, al menos, un elemento. Entonces la recorremos hasta llegar al último nodo que, como observamos más arriba, podemos identificar porque su puntero al siguiente es `NULL`. Una vez que encontramos el último nodo (apuntado por `aux`) simplemente asignamos a `aux->sig` la dirección del nuevo. Con esto, el nuevo nodo pasará a ser el último elemento de la lista.

Analicemos ahora el caso en el que `p` es `NULL`. En este caso entraremos por la parte izquierda del `if`, donde tenemos que hacer que el nuevo nodo sea también el primero asignando a `p` la dirección que contiene `nuevo`.

Hacer que `p` apunte al nuevo nodo implica modificar su valor. Recordemos que dentro de la función `agregar`, `p` es la dirección del puntero que apunta al primer nodo de la lista. Entonces `p*` es el puntero al primer nodo de la lista. Representaremos esta situación en la siguiente figura.

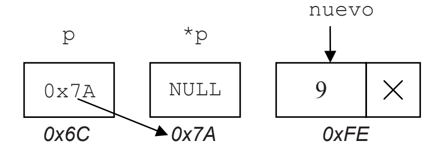


Fig. 11.8 Antes de asignar a `p` la dirección del nuevo nodo.

En la figura vemos que la variable `p` contiene la dirección de un espacio de memoria en el que actualmente se aloja el valor nulo `NULL`. Tenemos acceso a este espacio a través de `*p`.

Luego, para colocar al nuevo nodo como primer elemento de la lista tenemos que hacer que `*p` deje de tener `NULL` y pase a tener la dirección del nuevo elemento que, según la figura, es *0xFE*.

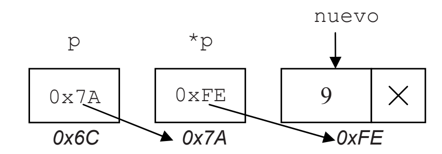


Fig. 11.9 Después de asignar a `p` la dirección del nuevo nodo.

Luego de la asignación `*p=nuevo` el nuevo nodo pasará a ser el primer y único elemento de la lista.

Veamos el código fuente de la función `agregar`.

```
void agregar(Nodo** p, int v)
{
   // creamos el nuevo nodo
   Nodo *nuevo = (Nodo*)malloc(sizeof(Nodo));
   nuevo->valor=v;
   nuevo->sig=NULL;

   // si la lista esta vacia entonces hacemos que p apunte al nuevo nodo
   if( *p==NULL )
   {
      *p=nuevo;
   }
   else
   {
      Nodo* aux=*p;

      // recorremos la lista hasta llegar al ultimo nodo
      while( aux->sig!=NULL )
      {
         // avanzamos a aux al proximo nodo
         aux=aux->sig;
      }

      // como aux apunta al ultimo entonces su siguiente sera el nuevo nodo
      aux->sig=nuevo;
   }
}
```

Ahora podemos hacer un programa en el que creamos una lista enlazada y mostramos su contenido.

```
int main()
{
   // inicializamos la lista
   Nodo* p=NULL;

   // le agregamos valores a traves de la funcion agregar
   agregar(&p,5);
   agregar(&p,8);
   agregar(&p,7);
   agregar(&p,3);
   agregar(&p,9);
```

```
    // mostramos por pantalla el contenido de lista
    // la funcion mostrar la analizaremos a continuacion
    mostrar(p);

    // antes de finalizar el programa liberamos la memoria
    // que ocupan los nodos de la lista (lo analizaremos mas adelante)
    liberar(&p);

    return 0;
}
```

Aquí definimos la variable `p` de tipo `Nodo*`. Durante el programa `p` será el puntero al primer nodo de la lista. Luego invocamos a la función `agregar` para agregarle elementos. Al final, invocamos a las funciones `mostrar` y `liberar` que analizaremos enseguida. La primera para mostrar por pantalla cada uno de los elementos de la lista y la segunda para liberar la memoria que ocupan.

11.5.2 Recorrer una lista para mostrar su contenido

Recorrer una lista para mostrar su contenido resulta muy fácil. Simplemente, tenemos que posicionarnos en el primer nodo, mostrar su valor y avanzar al siguiente. Si repetimos este proceso hasta llegar al final de la lista habremos resuelto nuestro problema.

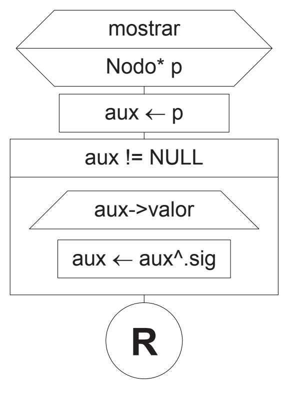


Fig. 11.10 Recorre la lista mostrando el valor de cada uno de sus nodos.

Notemos que en esta función simplemente recibimos el puntero al primer nodo de la lista (de tipo `Nodo*`), no su dirección (`Nodo**`). Esto es porque aquí no vamos a modificarlo. Simplemente, recorremos la lista pasando por cada uno de sus nodos para mostrar su valor por pantalla.

11.5.3 Liberar la memoria que utilizan los nodos de una lista enlazada

Como ya estudiamos, la memoria necesaria para alojar los elementos que agregamos a la lista se gestiona, dinámicamente, a través de la función de C `malloc`. La memoria gestionada por `malloc` es persistente y permanece “asignada” durante toda la ejecución del programa.

Debido a lo anterior, será nuestra responsabilidad liberar la memoria cuando ya no la necesitemos. Para esto, desarrollaremos la función `liberar` que recorre la lista enlazada liberando la memoria que ocupan cada uno de sus nodos.

Veamos el algoritmo de la función y luego lo analizaremos.

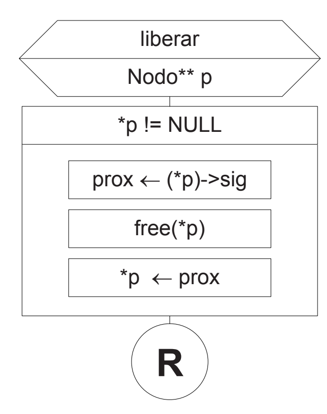


Fig. 11.11 Libera la memoria que utiliza una lista enlazada.

Esta función libera la memoria que utiliza la lista enlazada direccionada por `p` (puntero al primer nodo de la lista). Además, al finalizar, debe asignar el valor `NULL` a `p` ya que luego de liberar la memoria la lista quedará vacía. Esto significa que tenemos que recibir a `p` por referencia (`Nodo**`).

El algoritmo finalizará cuando `p` tenga el valor `NULL`. Esto, dentro de la función, se traduce como `while( *p!=NULL )`. Recordemos que recibimos a `p` por referencia; por lo tanto, la dirección del primer nodo de la lista es `*p`.

Como `*p` es la dirección del primer nodo entonces `(*p)->sig` es la dirección del segundo. La asignación `prox=(*p)->sig` asigna a la variable `prox` la dirección del segundo elemento de la lista. Los paréntesis son necesarios ya que el operador "flecha" tiene precedencia sobre el operador "asterisco". Si no utilizamos paréntesis entonces "estaríamos hablando del siguiente de `p`". Recordemos que, dentro de la función, `p` es de tipo `Nodo**`, no tiene campo `sig`. Quien tiene ese campo es `*p`.

Luego de la asignación `prox=(*p)->sig` podemos liberar la memoria direccionada por `*p` ya que la dirección del siguiente nodo está resguardada en `prox`.

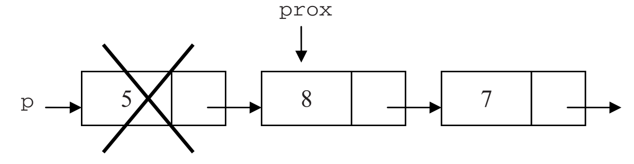


Fig. 11.12 Libera la memoria del primer nodo de la lista.

Ahora debemos hacer que el primer nodo de la lista sea el que está siendo apuntado por `prox`. La asignación `*p=prox` descarta, definitivamente, el primer nodo (ya liberado mediante la función `free`) y hace que la lista comience desde el segundo.

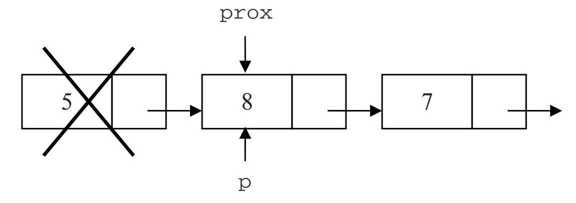


Fig. 11.13 La lista comienza desde el que hasta ahora era el segundo nodo.

Notemos que en la figura mantuvimos el dibujo del "viejo primer nodo de la lista" ya que el hecho de que lo hayamos liberado con `free` no implica que la información se haya perdido o borrado. Simplemente, ese espacio de memoria ya no pertenece más a nuestro programa y, de hecho, no tenemos forma de accederlo porque no lo tenemos apuntado con ningún puntero. Quedó desreferenciado.

Ahora sí, veamos el código de la función `liberar`.

```
void liberar(Nodo** p)
{
   while( *p!=NULL )
   {
      Nodo* prox=(*p)->sig;
      free(*p);
      *p=prox;
   }
}
```

Cuando `p` apunte al último nodo de la lista la asignación `prox=(*p)->sig` asignará el valor `NULL` a `prox`. Luego, al hacer `*p=prox` estaremos asignando `NULL` a `p` con lo que, finalmente, la lista quedará vacía y la memoria que ocupaba estará liberada.

11.5.4 Determinar si la lista contiene un valor determinado

Naturalmente, si tenemos una lista de elementos en algún momento vamos a necesitar determinar si esta contiene o no un cierto valor. Para esto, desarrollaremos la función `buscar` que realiza una búsqueda secuencial sobre los nodos de la lista hasta encontrar aquel nodo cuyo valor sea el que estamos buscando.

Veamos la siguiente lista enlazada y supongamos que queremos determinar si contiene un nodo cuyo valor sea 7.

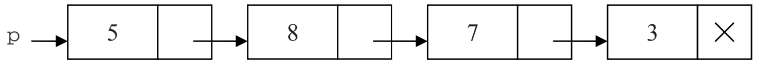


Fig. 11.14 Lista enlazada.

A simple vista, resulta obvio que sí, pero para desarrollar un algoritmo que lo pueda determinar tendremos que analizar el primer nodo y, si no contiene el valor que estamos buscando, pasar al siguiente y así hasta analizar el último elemento de la lista.

Utilizaremos un puntero auxiliar `aux`, inicialmente, apuntando al primer nodo. Si el nodo apuntado por `aux` no tiene el valor que estamos buscando entonces lo haremos avanzar para que apunte al siguiente.

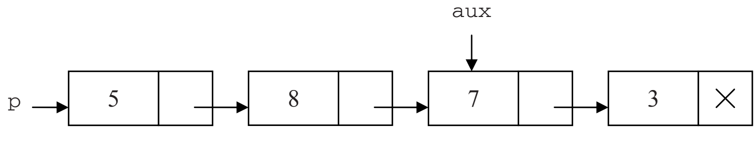


Fig. 11.15 Encontramos el nodo que buscábamos.

Si el valor de alguno de los nodos de la lista coincide con el que estamos buscando entonces podemos dar por finalizado el algoritmo.

La función retornará un puntero al nodo cuyo valor sea el que buscamos.

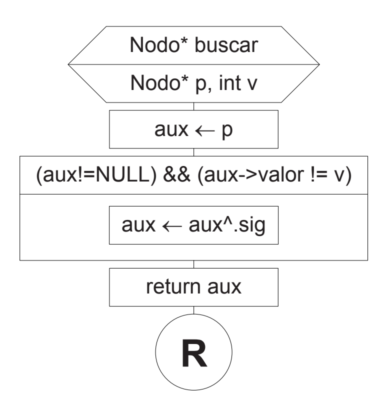


Fig. 11.16 Búsqueda secuencial sobre los nodos de una lista enlazada.

Ahora bien, ¿qué sucederá cuándo busquemos un elemento que no esté en la lista, por ejemplo el 25?

Veamos: luego de analizar y avanzar sobre todos los nodos de la lista, al llegar al último observaremos la siguiente situación:

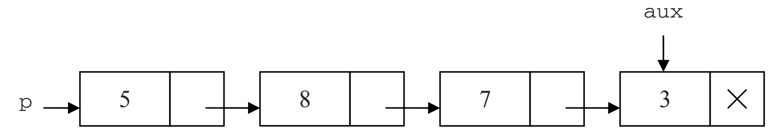


Fig. 11.17 Llegamos al último nodo sin encontrar lo que buscamos.

En este caso, el nodo apuntado por `aux` tiene el valor 3, que no coincide con el valor 25 que estamos buscando. Luego, al avanzar el puntero haremos `aux=aux->sig`. Como `aux` apuntaba al último nodo de la lista resulta que su referencia al siguiente es `NULL`; por lo tanto, le estaremos asignando el valor `NULL` a `aux`, lo que hará finalizar el ciclo iterativo y, al retornar `aux`, estaremos retornando `NULL`.

Luego, la función `buscar` retorna un puntero al nodo que contiene el valor que buscamos o `NULL` si ningún nodo de la lista contiene dicho valor.

```
Nodo* buscar(Nodo* p,int v)
{
   Nodo* aux=p;
   while( (aux!=NULL) && (aux->valor!=v) )
   {
      aux=aux->sig;
   }

   return aux;
}
```

Antes de continuar analizando más operaciones sobre listas vamos a desarrollar un pequeño programa cuya resolución requerirá usar las funciones que estudiamos.

Problema 11.1

Se ingresa por teclado un conjunto de valores enteros. El ingreso de datos finalizará cuando el usuario ingrese el valor 0 (cero). Luego se ingresa otro conjunto de valores enteros y, por cada uno de estos, se debe informar si el valor ingresado pertenece o no al primer conjunto.

Análisis

Este ejercicio es prácticamente idéntico a alguno de los que analizamos en el capítulo de *arrays*. La diferencia es que aquí no sabemos cuántos elementos tiene el primer conjunto que ingresará el usuario, por lo que no podemos utilizar un *array* para guardarlo. Utilizaremos una lista enlazada para mantener todos los valores del primer conjunto y luego, por cada valor del segundo conjunto, buscaremos en la lista para determinar si ese valor fue o no ingresado como parte del primero.

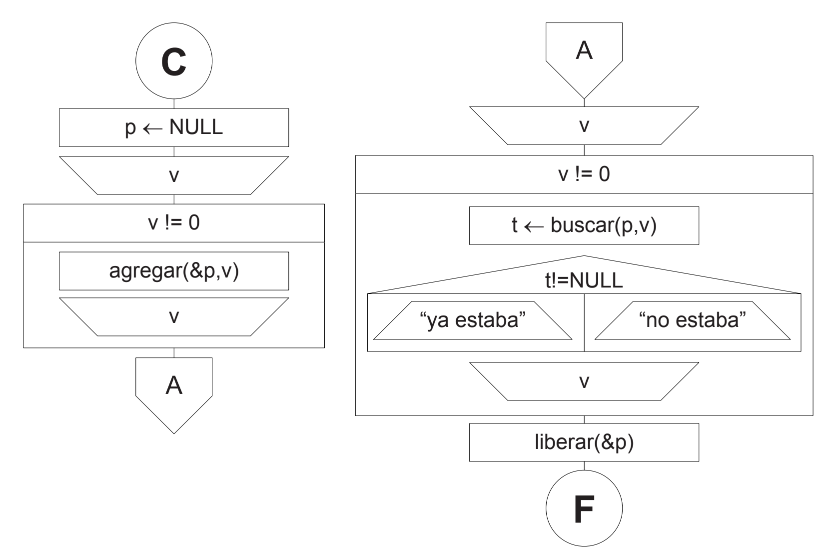


Fig. 11.18 Determina qué elementos del segundo conjunto pertenecen también al primero.

Como vemos, gracias a las operaciones `agregar`, `buscar` y `liberar` pudimos usar una lista enlazada abstrayéndonos completamente de toda la complejidad que implica su implementación.

Se le recomienda al lector detener momentáneamente la lectura para extender el ejercicio anterior resolviendo los siguientes puntos:

1. De los elementos del segundo conjunto que no estén contenidos en el primero informar:
    a. ¿Cuántos son pares?
    b. Promedio de los valores positivos.
3. Cantidad de elementos del primer conjunto.
4. Llamemos A al primer conjunto y B al segundo. Mostrar por pantalla todos los elementos del conjunto C, siendo C=A-B.

11.5.5 Eliminar un elemento de la lista

El siguiente algoritmo nos permitirá eliminar un nodo de la lista cuyo valor coincida con el que especifiquemos como argumento.

Dada la siguiente lista enlazada, supongamos que queremos eliminar el nodo cuyo valor es 7.

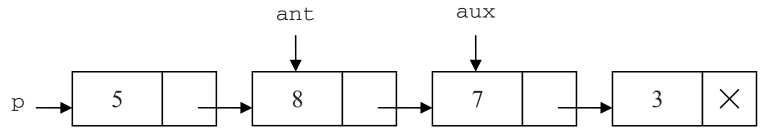


Fig. 11.19 Lista enlazada sobre la que vamos a eliminar un nodo.

Para eliminar el elemento cuyo valor es 7 tenemos que obtener dos punteros: uno que apunte al nodo que vamos a eliminar (`aux`) y otro que apunte al nodo anterior (`ant`).

Una vez logrado esto eliminaremos el nodo que está siendo referenciado por `aux` haciendo que el siguiente de `ant` apunte al siguiente de `aux`.

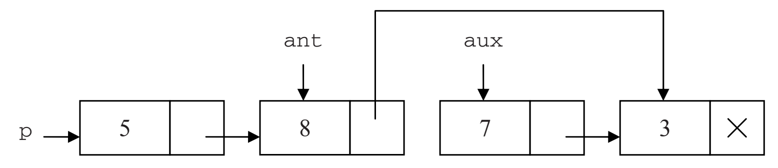


Fig. 11.20 Sacamos de la lista al nodo referenciado por `aux`.

Luego de hacer que el siguiente de `ant` apunte al siguiente de `aux`, el nodo referenciado por `aux` dejó de ser parte de la lista, pero aún ocupa memoria. El próximo paso será liberar ese espacio de memoria haciendo `free(aux)`.

Para obtener la referencia al nodo que vamos a eliminar y la referencia al nodo anterior recorreremos secuencialmente la lista y, antes de avanzar al siguiente nodo, asignaremos en el puntero `ant` el valor actual de `aux`.

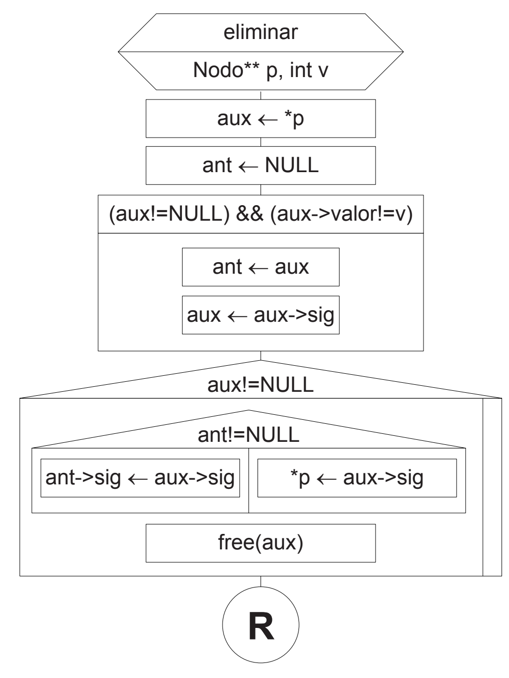


Fig. 11.21 Elimina un elemento de la lista.

Notemos que la función recibe a `p` por referencia porque en el caso de eliminar al primer nodo `p` debe pasar a apuntar al siguiente y si la lista tiene un único nodo entonces `p` debe quedar en `NULL`.

```
void eliminar(Nodo** p, int v)
{
   Nodo* aux=*p;
   Nodo* ant=NULL;

   while( (aux!=NULL) && (aux->valor!=v) )
   {
      ant=aux;
      aux=aux->sig;
   }

   if( aux!=NULL )
   {
      if( ant!=NULL )
      {
         ant->sig=aux->sig;
      }
      else
      {
         *p=aux->sig;
      }

      free(aux);
   }
}
```

Problema 11.2

Utilizando la función `eliminar` fácilmente podemos resolver el punto 3 del ejercicio planteado más arriba donde se pedía ingresar dos conjuntos A y B para mostrar el conjunto resultante C=A-B.

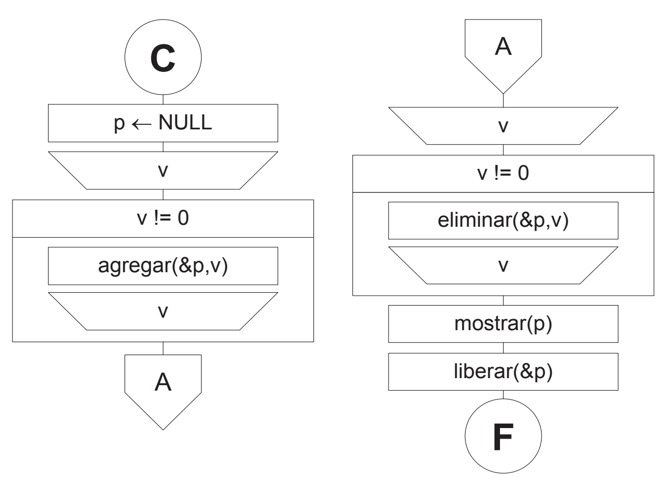


Fig. 11.22 Elimina elementos de una lista enlazada.

La lógica es muy simple: agregamos todos los valores del conjunto A a la lista referenciada por `p` usando la función `agregar`. Luego con la función `eliminar` eliminamos de la lista cada uno de los valores del conjunto B. Finalmente, con la función `mostrar` mostramos la lista cuyos elementos serán todos los valores de A que no formen parte del conjunto B.

11.5.6 Insertar un valor respetando el ordenamiento de la lista

Analizaremos ahora un algoritmo que nos permita insertar un valor dentro de una lista enlazada de forma tal que su ubicación respete el orden natural de esta. Veamos un ejemplo.

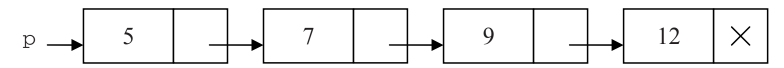


Fig. 11.23 Lista enlazada ordenada.

Los elementos representados en los nodos de esta lista se encuentran ordenados naturalmente. Para insertar un nuevo nodo, por ejemplo, con valor 10 tenemos que recorrer la lista hasta encontrar el primer elemento mayor que, en este caso, es 12 y quedarnos con un puntero al elemento anterior (9).

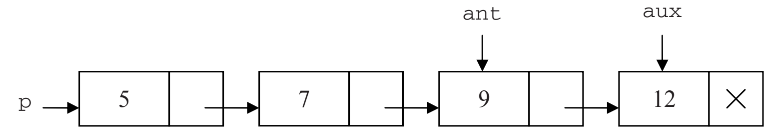


Fig. 11.24 Lista enlazada ordenada, a punto de insertar un nuevo valor.

Luego, para insertar el nodo con valor 10 en la posición que corresponda debemos hacer que el siguiente del nuevo nodo apunte a `aux` y también que el siguiente de `ant` apunte al nuevo nodo.

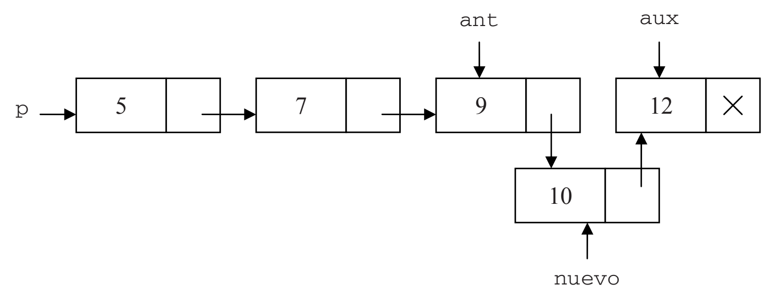


Fig. 11.25 Lista enlazada ordenada, con nuevo valor.

Tenemos que analizar los siguientes casos particulares:

El primero se da cuando el elemento que vamos a insertar es menor que el primer elemento de la lista. En este caso tenemos que modificar el valor de `p` para hacerlo apuntar al nuevo nodo; el siguiente del nuevo debe apuntar al viejo valor de `p`.
Supongamos que vamos a insertar el valor 3, entonces:

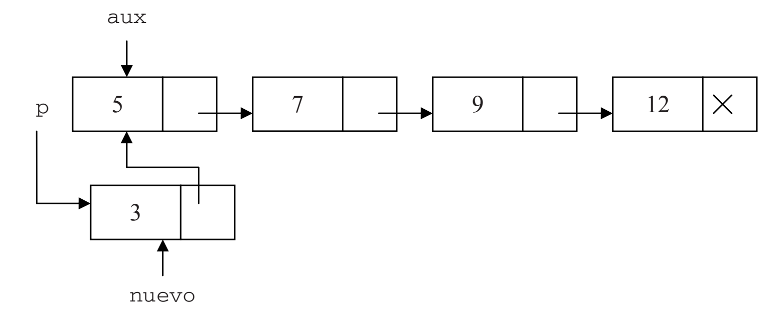


Fig. 11.26 Lista enlazada ordenada, con nuevo valor al principio.

Como `aux` quedó apuntando al primer nodo de la lista, entonces el puntero al nodo anterior habrá quedado en `NULL`.

El otro caso particular se da cuando el elemento que vamos a insertar es mayor que todos los elementos de la lista, por lo que tendremos que ubicarlo al final. En este caso, `ant` quedará apuntando al último nodo y `aux` habrá quedado en `NULL`.

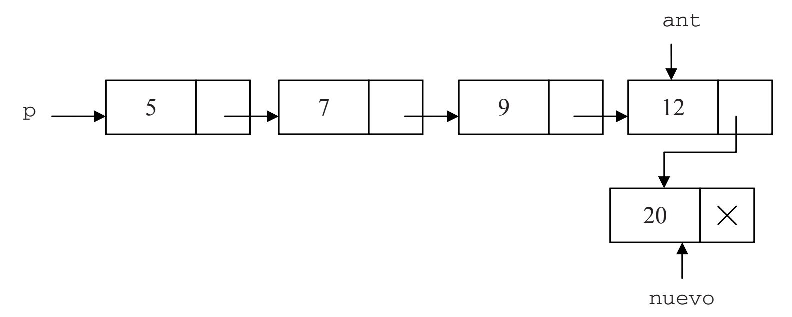


Fig. 11.27 Lista enlazada ordenada, con nuevo valor al final.

Veamos el algoritmo.

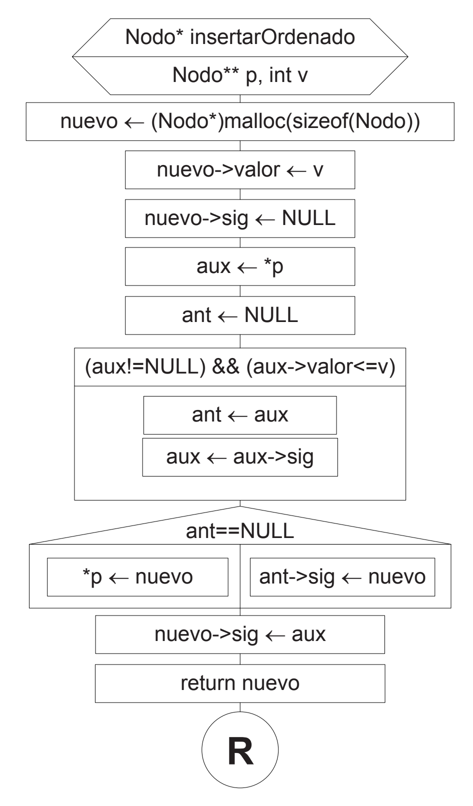


Fig. 11.28 Insertar un valor respetando el orden natural de la lista enlazada.

```
Nodo* insertarOrdenado(Nodo** p, int v)
{
   Nodo* nuevo = (Nodo*) malloc(sizeof(Nodo));
   nuevo->valor = v;
   nuevo->sig = NULL;

   Nodo* aux = *p;
   Nodo* ant = NULL;

   while( (aux != NULL) && (aux->valor <= v) )
   {
      ant = aux;
      aux = aux->sig;
   }

   if( ant == NULL )
   {
      *p = nuevo;
   }
   else
   {
      ant->sig = nuevo;
   }

   nuevo->sig = aux;
   return nuevo;
}
```

11.5.7 Insertar un valor solo si la lista aún no lo contiene

Para terminar desarrollaremos un algoritmo que nos permitirá insertar (en orden) un valor solo si la lista aún no lo contiene. Lo implementaremos como una función que retornará un puntero al nodo que contiene dicho valor, si la lista ya lo contenía. En caso contrario, insertará el valor y retornará un puntero al nodo recientemente insertado.

Para determinar si el valor `v` ya estaba contenido o se agregó luego de haber invocado a la función, recibiremos el parámetro por referencia `enc` (encontrado) donde asignaremos *true* o *false* según corresponda. Veamos la implementación.

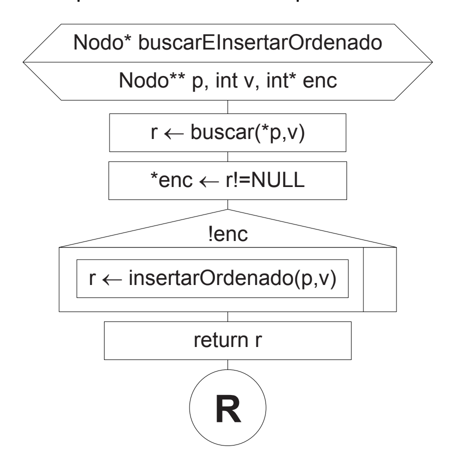


Fig. 11.29 Inserta un valor en la lista solo si aún no lo contiene.

Como ya tenemos programadas las funciones `buscar` e `insertar` la implementación de este algoritmo resulta ser trivial.

```
Nodo* buscarEInsertarOrdenado(Nodo** p, int v, int* enc)
{
   Nodo* r = buscar(*p,v);
   *enc = r!=NULL;

   if( !*enc )
   {
      r = insertarOrdenado(p,v);
   }

   return r;
}
```

11.6 Estructura Pila (LIFO)

La pila (*stack*) es una estructura lineal sobre la que rigen ciertas restricciones a la hora de agregar o quitar elementos.

A diferencia de las listas enlazadas en las que desarrollamos operaciones para insertar, agregar y eliminar nodos sin ningún tipo de limitación, decimos que la pila es una estructura lineal restrictiva de tipo LIFO (*Last In First Out*). Esto indica que el último elemento que ingresó a la pila debe ser el primero en salir.

Fácilmente, podemos visualizar una pila si pensamos, por ejemplo, en una pila de libros. Supongamos que tenemos varios libros desparramados sobre una mesa. Tomamos uno (lo llamaremos A) y lo colocamos frente a nosotros. Luego tomamos otro (lo llamaremos B) y lo colocamos encima de A. Luego tomamos otro libro más (lo llamaremos C) y lo colocamos encima de B. Así conformamos una pila de libros y en la cima de la pila siempre quedará ubicado el último libro que apilamos. Si queremos sacar un elemento de la pila de libros tomaremos siempre el que está en la cima. Así, el primer libro que sacaremos será C (el último que apilamos). El siguiente libro que tomaremos será B (el anteúltimo que apilamos) y por último tomaremos el libro A (el primero).

11.6.1 Implementación de la estructura pila

Implementaremos la pila sobre una lista enlazada para la cual solo desarrollaremos dos operaciones: `poner` (apilar un elemento) y `sacar` (obtener y eliminar un elemento de la pila).

11.6.2 Operaciones `poner` (push) y `sacar` (pop)

Supongamos que queremos poner en una pila los elementos del siguiente conjunto: {3, 2, 1}. Para esto, definimos un puntero `p` de tipo `Nodo*` inicializado en `NULL` y luego agregamos cada uno de los elementos del conjunto al inicio de la lista apuntada por `p`.

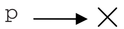


Fig. 11.30 Pila implementada sobre una lista enlazada, aún sin elementos.

Agregamos el 3 (primer elemento del conjunto):

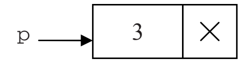


Fig. 11.31 Pila implementada sobre una lista enlazada con un único elemento.

Agregamos el 2 (segundo elemento del conjunto):

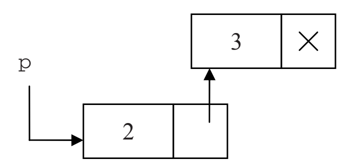


Fig. 11.32 Pila implementada sobre una lista enlazada con dos elementos.

Agregamos el 1 (último elemento del conjunto):

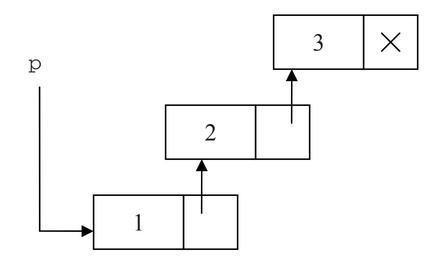


Fig. 11.33 Pila implementada sobre una lista enlazada con tres elementos

Luego, para sacar un elemento de la pila siempre tomaremos el primero de la lista.

Veamos el diagrama y la codificación de las funciones `poner` y `sacar` que, respectivamente, permiten apilar y desapilar elementos en una pila de enteros.

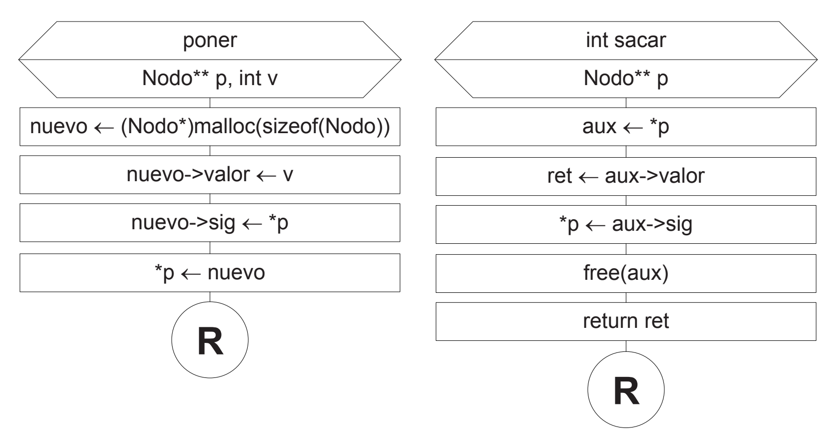


Fig. 11.34 Operaciones poner y sacar.

```
void poner(Nodo** p, int v)
{
   Nodo* nuevo = (Nodo*) malloc(sizeof(Nodo));
   nuevo->valor = v;
   nuevo->sig = *p;
   *p = nuevo;
}

int sacar(Nodo** p)
{
   Nodo* aux=*p;
   int ret=aux->valor;

   *p=aux->sig;
   free(aux);

   return ret;
}
```

En muchos libros y documentos que circulan por Internet, es habitual encontrar que los autores se refieran a las operaciones `poner` y `sacar` como *push* y *pop* respectivamente.

Problema 11.3

Se ingresa por teclado un conjunto de valores que finaliza con la llegada de un 0 (cero). Se pide mostrar los elementos del conjunto en orden inverso al original.

Para resolver este programa utilizaremos una pila en la que apilaremos los valores a medida que el usuario los vaya ingresando. Luego, mientras que la pila no esté vacía, sacaremos uno a uno sus elementos para mostrarlos por pantalla.

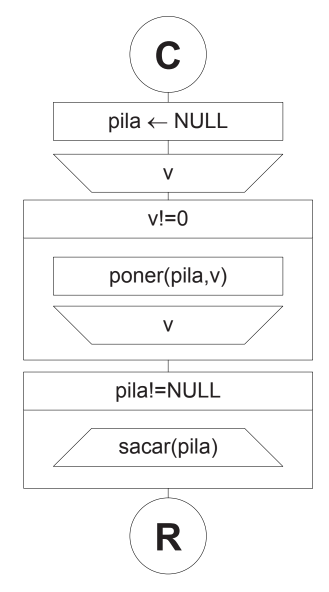


Fig. 11.35 Muestra un conjunto de valores en orden inverso.

Notemos que al finalizar el programa no es necesario liberar la memoria ocupada por la pila porque la función `sacar`, además de retornar el valor del elemento ubicado en la cima, desenlaza el nodo y lo libera con la función `free` de C.

11.6.3 Determinar si la pila tiene elementos o no

Algunos autores definen también la operación `pilaVacia` que retorna *true* o *false* según la pila tenga o no elementos apilados. Según mi criterio, esta operación es innecesaria ya que la pila estará vacía cuando el puntero al primer nodo de la lista sea `NULL`.

De cualquier manera, podemos desarrollar esta operación en una única línea de código como vemos a continuación:

```
int pilaVacia(Nodo* p)
{
   return p==NULL;
}
```

Recordemos que en C los valores booleanos se manejan con valores enteros. Así, el número 0 (cero) equivale al valor booleano *false* y el número 1 (uno) o cualquier otro valor diferente de 0 equivale a *true*.

Utilizando la función `pilaVacia` el último `while` del diagrama anterior quedaría así:

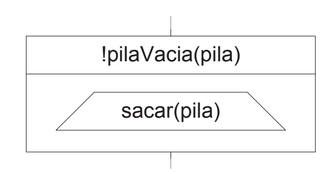


Fig. 11.36 Mientras la pila no esté vacía, saca y muestra cada uno de sus elementos.

11.7 Estructura Cola (FIFO)

La cola (*queue*) es también una estructura lineal restrictiva en la que solo podremos poner y sacar elementos respetando la siguiente restricción: el primer elemento en llegar a la cola será también el primero en salir (FIFO, *First In First Out*).

Para comprender este concepto, basta con imaginar la cola que se forma en la caja de un supermercado donde la cajera atiende a los clientes que forman la cola respetando el orden de llegada; por lo tanto, el primero que llegó también será el primero en ser atendido y en salir.

Solo a título informativo, se le sugiere al lector pensar en la cola que se forma en una caja de supermercado con prioridad para mujeres embarazadas. En general, en este tipo de cajas también se admiten hombres y mujeres no embarazadas que serán atendidos según el orden de llegada. Sin embargo, si llega una mujer embarazada la cajera le dará el primer lugar y la atenderá inmediatamente. A este tipo de colas, se las llama “colas jerarquizadas” o “colas por prioridad” y son objeto de estudio en los libros de sistemas operativos.

Así como implementamos la pila sobre una lista enlazada, la cola la implementaremos sobre un caso particular de lista enlazada: la lista enlazada circular.

11.7.1 Lista enlazada circular

Una lista circular es una lista enlazada en la que el último nodo apunta al primero. Veamos gráficamente tres casos: una lista circular vacía, una lista circular con un único elemento y una lista circular con más de un elemento.

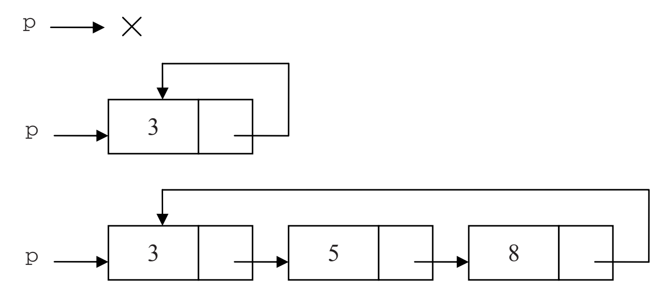
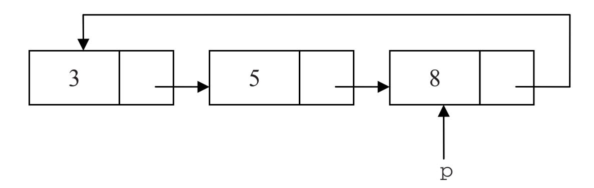


Fig. 11.37 Lista enlazada circular.

Dado que en una lista circular el último nodo tiene una referencia al primero resultará más provechoso mantener a `p` apuntando al último nodo de la lista ya que desde allí podremos acceder al primero, que estará referenciado por `p->sig`.



Así, en una lista circular siempre tenemos referenciado el último nodo que agregamos (`p`) e, indirectamente, también tenemos referenciado el primero (`p->sig`).

11.7.2 Implementar una cola sobre una lista circular

Volviendo al ejemplo de la caja en el supermercado, podemos ver que la gente que llega a la cola se ubica al final. En cambio, la cajera siempre atiende al primero, que luego se retira de la cola para dejar su lugar al segundo y así sucesivamente.

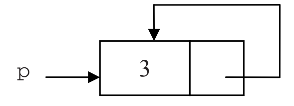
El análisis anterior demuestra que, para la implementación de la estructura cola, necesitamos una lista enlazada con dos punteros: uno al primer nodo y uno al último. Justamente, esta característica la encontramos en una lista circular.

Luego, para encolar un elemento lo agregaremos al final de la lista y para desencolar eliminaremos al primero de la lista.

Analicemos una cola inicialmente vacía.



Ahora encolamos el valor 3:



Encolemos el valor 5 y luego el 8:

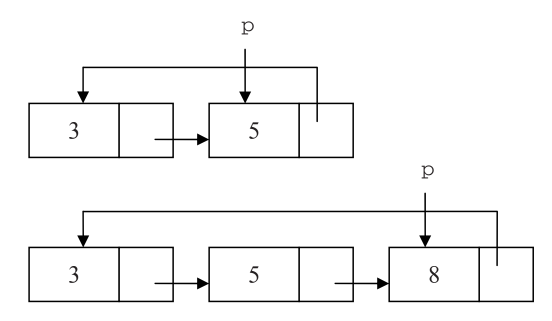


Fig. 11.38 Estructura Cola implementada sobre una lista circular.

Como vemos, el final de la lista siempre está apuntado por `p` y el inicio está referenciado por el `p->sig`.

El primer valor que encolamos es el 3 y es el que debe salir si queremos desencolar un elemento. Para esto, según nuestro ejemplo, tenemos que hacer que el siguiente de 8 apunte al siguiente de 3 (es decir, al 5). Si con una variable `aux` apuntamos al siguiente de `p` entonces para desencolar simplemente hacemos que el siguiente de `p` apunte al siguiente de `aux` y luego liberamos `aux`.

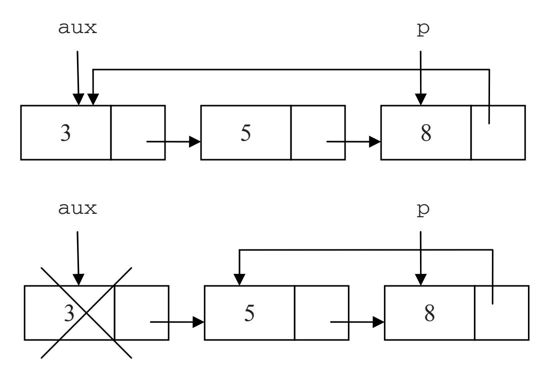


Fig. 11.39 Desencolar un elemento.

11.7.3 Operaciones `encolar` y `desencolar`

Veamos el diagrama de la función `encolar` que agrega un elemento al final de la lista circular direccionada por `p`.

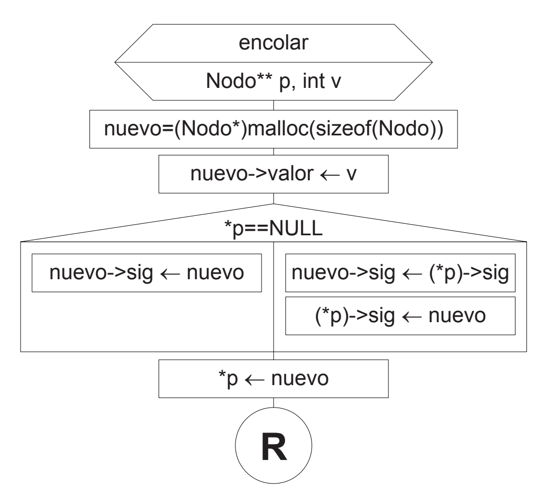


Fig. 11.40 Encola un elemento agregándolo al final de la lista circular.

```
void encolar(Nodo** p, int v)
{
   Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
   nuevo->valor=v;

   if( *p==NULL )
   {
      nuevo->sig=nuevo;
   }
   else
   {
      nuevo->sig=(*p)->sig;
        (*p)->sig=nuevo;
   }

   *p=nuevo;
}
```

Veamos ahora cómo desencolar considerando el caso particular que se da al desencolar el último elemento de la cola. Esta situación la identificamos cuando resulta que `(*p)->sig` es igual a `p`.

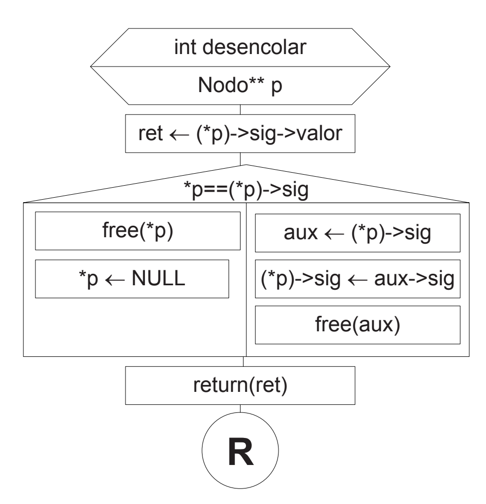


Fig. 11.41 Desencola un elemento tomándolo del principio de la lista circular.

```
int desencolar(Nodo** p)
{
   int ret=(*p)->sig->valor;

   if( *p==(*p)->sig )
   {
      free(*p);
      *p=NULL;
   }
   else
   {
      Nodo* aux = (*p)->sig;
      (*p)->sig=aux->sig;
      free(aux);
   }

   return ret;
}
```

Veamos ahora el siguiente programa:

```
int main()
{
   Nodo* p=NULL;

   // encolamos varios elementos
   encolar(&p,1);
   encolar(&p,2);
   encolar(&p,3);

   // desencolamos un elemento (sale el 1)
   printf("%d\n",desencolar(&p));
```

```
   // desencolamos un elemento (sale el 2)
   printf("%d\n",desencolar(&p));

   // encolamos mas elementos
   encolar(&p,4);
   encolar(&p,5);
   encolar(&p,6);

   // desencolamos un elemento (sale el 3)
   printf("%d\n",desencolar(&p));

   // desencolamos mientras queden elementos en la cola
   while( p!=NULL )
   {
      printf("%d\n",desencolar(&p));
   }

   return 0;
}
```

La salida será:

```
1
2
3
4
5
6
```

11.8 Lista doblemente enlazada

Así como en una lista enlazada cada nodo tiene un puntero al siguiente, en una lista doblemente enlazada cada nodo tiene dos punteros: uno apuntando al siguiente y otro apuntando al anterior.

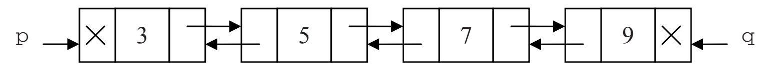


Fig. 11.42 Lista doblemente enlazada.

En la figura vemos una lista doblemente enlazada: cada nodo tiene un puntero al siguiente y un puntero al anterior. Obviamente, el puntero al anterior del primer nodo y el puntero al siguiente del último son `NULL`. También vemos que manejamos dos punteros `p` y `q` que apuntan al primer y al último nodo de la lista respectivamente.

La lista doblemente enlazada utiliza más memoria que la lista enlazada, pero ofrece las siguientes ventajas:

- La lista puede recorrerse en ambas direcciones.
- Las operaciones `insertar` y `eliminar` utilizan menor cantidad de instrucciones ya que el mismo nodo tiene la dirección del siguiente y del anterior.

Dejamos a cargo del lector el desarrollo de las siguientes operaciones que aplican sobre las listas doblemente enlazadas: `buscar`, `agregarAlFinal`, `agregarAlPrincipio`, `insertarOrdenado`, `eliminar`, `buscarEInsertarOrdenado`.

11.9 Nodos que contienen múltiples datos

Hasta aquí, por cuestiones de simplicidad, solo trabajamos con nodos que permitían guardar un único valor entero.

```
typedef struct Nodo
{
   int v;            // valor o informacion que guarda el nodo
   struct Nodo* sig; // referencia al siguiente nodo de la lista
}Nodo;
```

Sin embargo, es probable que necesitemos guardar más datos dentro de un mismo nodo para implementar, por ejemplo, una pila de personas. Esta situación se puede resolver de dos maneras, como veremos más abajo, luego de que recordemos la estructura `Persona`.

```
typedef struct Persona
{
   int dni;
   char nombre[20];
   long fechaNac;
}Persona;
```

11.9.1 Nodo con múltiples campos

La primera solución para este problema consiste en agregar al nodo todos los campos de la estructura que queremos guardar. En el caso de `struct Persona` el nodo quedaría así:

```
typedef struct Nodo
{
   int dni;          // datos de la persona
   char nombre[20];  // datos de la persona
   long fechaNac;    // datos de la persona
   struct Nodo* sig; // referencia al siguiente nodo
}Nodo;
```

Para implementar una pila con nodos de este tipo, las operaciones `poner` y `sacar` deberían plantearse de la siguiente manera:

```
void poner(Nodo** p, Persona v)
{
   Nodo* nuevo = (Nodo*)malloc(sizeof(Persona));

   // asignamos uno a uno los campos de v a nuevo
   nuevo->dni = v.dni;
   nuevo->fechaNac=v.fechaNac;
   strcpy(nuevo->nombre,v.nombre);

   // ahora si, enlazamos el nodo al principio de la lista
   nuevo->sig=*p;
   *p=nuevo;
}
```

```
Persona sacar(Nodo** p)
{
   // definimos una variable de tipo persona
   Persona v;

   // asignamos uno a uno los campos
   v.dni = (*p)->dni;
   v.fechaNac = (*p)->fechaNac;
   strcpy(v.nombre,(*p)->nombre);

   // ahora desenlazamos el primer nodo
   Nodo* aux = *p;
   *p = (*p)->sig;
   free(aux);

   return v;
}
```

11.9.2 Nodo con un único valor de tipo `struct`

La otra opción para que un nodo permita guardar varios valores es definirle un único campo de tipo `struct`. Con esta alternativa podemos implementar una pila de personas de la siguiente manera: Primero el nodo:

```
typedef struct Nodo
{
   Persona info;      // todos los datos de la persona
   struct Nodo* sig; // referencia al siguiente nodo
}Nodo;
```

Ahora las operaciones `poner` y `sacar`.

```
void poner(Nodo** p, Persona v)
{
   Nodo* nuevo = (Nodo*)malloc(sizeof(Persona));

   // asignamos los datos de la persona
   nuevo->info = v;

   // ahora si, enlazamos el nodo al principio de la lista
   nuevo->sig=*p;
   *p=nuevo;
}

Persona sacar(Nodo** p)
{
   // rescatamos los datos del primer nodo
   Persona v = (*p)->info;

   // ahora desenlazamos el primer nodo
   Nodo* aux = *p;
   *p = (*p)->sig;
   free(aux);

   return v;
}
```

Si bien las dos soluciones son correctas, es evidente que la segunda opción implica menos trabajo y es más fácil de implementar.

Si en lugar de implementar una pila quisiéramos implementar una cola de personas deberíamos plantear las operaciones `encolar` y `desencolar` de la siguiente forma:

```
void encolar(Nodo** p, Persona v)
{
   // :
}

Persona desencolar(Nodo** p)
{
   // :
}
```

En el caso de implementar una lista de personas, la operación `agregar` se vería así:

```
void agregar(Nodo** p, Persona v)
{
   // :
}
```

Respecto de función `buscar` podríamos implementarla de dos maneras. Veamos:

```
Nodo* buscar(Nodo** p, int dni)
{
   // :
}
```

En esta implementación recibimos (por referencia) el puntero al primer nodo de la lista y el DNI de la persona que queremos buscar. Esto es correcto si el DNI es suficiente para identificar a una persona. Sin embargo, si necesitamos más datos o todos los datos de la estructura para identificar a una persona entonces el prototipo de la función `buscar` debería ser el siguiente:

```
Nodo* buscar(Nodo** p, Persona v)
{
   // :
}
```

En todos los casos la función `buscar` retorna un `Nodo*` ya que si encuentra a la persona que estamos buscando retornará un puntero al nodo que contiene sus datos, pero si ningún nodo tiene la información de dicha persona entonces retornará `NULL`.

11.10 Estructuras de datos combinadas

En la vida real, es probable que para resolver cierto tipo de problemas necesitemos pensar en estructuras mixtas que surjan al combinar algunas de las estructuras dinámicas que estudiamos en este capítulo y combinaciones entre estas y estructuras de datos estáticas como *arrays*, *structs*, etcétera.

11.10.1 Lista y sublista

Cuando los nodos de una lista enlazada, además de tener la información propia del nodo y la referencia al siguiente nodo de la lista, tienen un puntero al primer nodo de otra lista enlazada de nodos de otro tipo de datos, hablamos de "una lista principal y una lista secundaria" o simplemente de "lista y sublista".

Es fundamental el hecho de que los nodos de la lista principal y los nodos de la lista secundaria sean de diferentes tipos de datos ya que, si fuesen nodos del mismo tipo, entonces la estructura tomaría forma de árbol binario, tema que estudiaremos en el Capítulo 16.

A continuación, analizaremos un problema cuya resolución requiere del uso de una estructura de tipo "lista y sublista".

Problema 11.4

Se desea procesar un archivo que contiene información sobre los siniestros denunciados por los clientes de una compañía de seguros. En el archivo existe un registro por cada denuncia, con los siguientes campos:

- Número de matrícula – 6 caracteres
- Fecha – 10 caracteres con formato "*dd/mm/aaaa*"
- Detalles – 256 caracteres

No conocemos cuántas matrículas diferentes existen ni tampoco la cantidad máxima de siniestros denunciados que puede haber por cada una de ellas. Los registros en el archivo están ordenados ascendentemente por *fecha*.

Se pide emitir un listado ordenado por número de matrícula informando, por cada una, la fecha y el detalle de cada uno de los siniestros denunciados.

Análisis

El problema se resuelve creando una lista ordenada de matrículas donde cada nodo tiene un puntero que hace referencia a una lista de siniestros.

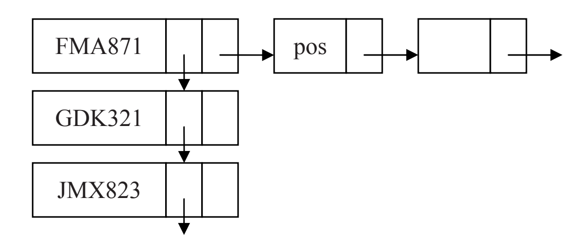


Fig. 11.43 Estructura combinada - lista y sublista.

En la figura vemos una lista enlazada de matrículas donde cada nodo, además de tener la referencia al siguiente nodo de la lista, tiene un puntero a una sublista de siniestros en la que cada nodo guarda la posición del registro del archivo que representa al siniestro.

Con esta estructura, la estrategia de resolución consiste en recorrer el archivo y, por cada registro, *buscar e insertar* en la lista principal y luego agregar un nodo al final de la sublista con la posición que ocupa el registro dentro del archivo.

Al finalizar el proceso podremos emitir el listado que nos piden recorriendo la lista principal y, por cada nodo, recorriendo la sublista y, por cada nodo, haciendo un acceso directo al archivo para obtener la fecha y el detalle del siniestro.

La definición de tipos para esta estructura de datos es la siguiente:

```
// estructura del nodo de la sublista de siniestros
typedef struct NSinies
{
   int pos;
   struct NSinies* sig;
}NSinies;

// estructura del nodo de la lista de matriculas
typedef struct NMat
{
   char nmat[6+1];       // 6 para la matricula + 1 para el '\0'
   struct NMat* sig;     // puntero al siguiente nodo
   struct NSinies* slst; // puntero al primer nodo de la sublista
}NMat;

// estructura del registro del archivo
typedef struct RSinies
{
   char nmat[6+1];
   char fecha[10+1];
   char detalles[256];
}RSinies;
```

En el programa principal, recorremos el archivo y, por cada registro, agregamos (sin repetición) un nodo en la lista de matrículas (lista principal) y agregamos también un nodo al final de la sublista de siniestros de esta matrícula.

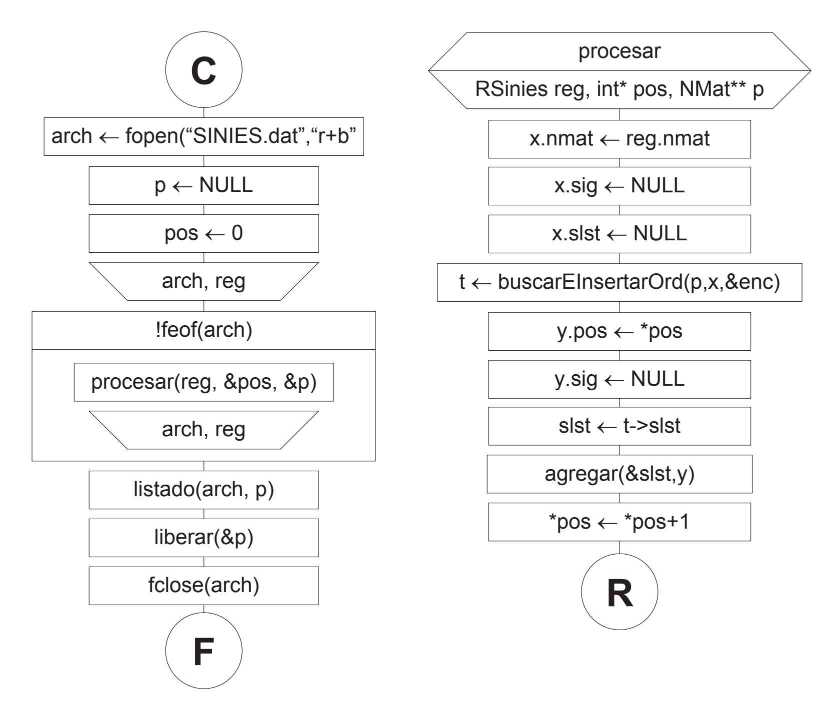


Fig. 11.44 Programa principal y función `procesar`.

Para emitir el listado recorremos la lista de matrículas y, por cada una, recorremos la sublista de siniestros. Por cada siniestro accedemos al archivo para recuperar la fecha y los detalles y mostramos la información.

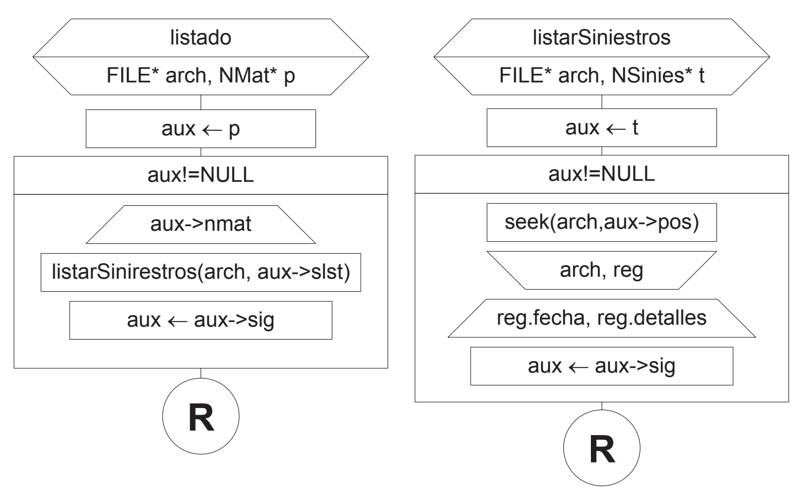


Fig. 11.45 Listado de matriculas y siniestros.

El listado que emitiremos estará ordenado por matrícula y, por cada matrícula, sus siniestros estarán ordenados por fecha ascendente porque este es el orden en el que se encuentran los registros del archivo.

11.10.2 Arrays de colas

Para resolver el siguiente problema utilizaremos *arrays* de punteros para implementar colas de espera.

Problema 11.5

Una empresa que brinda auxilio mecánico cubre 20 zonas, numeradas de 0 a 19. Cuando un abonado necesita ayuda se comunica telefónicamente, generándose así un nuevo caso por resolver.

Para abrir un caso, el abonado debe informarle al operador los siguientes datos:

- Número de abonado (8 dígitos)
- Número de zona (0 a 19)
- Dirección exacta en donde debe ser auxiliado (hasta 256 caracteres)

Diremos que un caso está "abierto" mientras no se le haya asignado un móvil de auxilio. Una vez establecida la asignación de un móvil a un caso, diremos que el caso está "asignado". Por último, el caso estará "cerrado" cuando el móvil que le fuera asignado reporte por radio al operador los siguientes datos.

- Número de caso (8 dígitos)
- Tipo de problema (3 dígitos)

En cada zona operan varios móviles, todos exclusivos de la zona. La información de estos móviles se encuentra en el archivo `MOVILES.dat`, donde cada registro tiene:

- Número de móvil (3 dígitos)
- Número de zona (0 a 19)

La asignación de casos debe ser uniforme. Es decir, a medida que los móviles de una zona se vayan liberando deberán formar una cola de espera para recibir un nuevo caso. Análogamente, los abonados serán atendidos según la zona en la que se encuentren y según el orden en el que fueron llamando para solicitar auxilio.

Se dispone de la función `horaActual` que retorna la hora del sistema expresada como un valor numérico entero con el siguiente formato: *hhmm*.

Se pide desarrollar un programa que automatice la asignación de móviles a los diferentes casos, según las pautas descritas más arriba. Al finalizar, se pide generar el archivo `CASOS.dat` con los siguientes datos:

- Número de caso
- Número de abonado
- Número de zona
- Número de móvil
- Tipo de problema
- Hora de apertura
- Hora de cierre

Análisis

El enunciado describe una operatoria en la que el usuario (el operador) está constantemente interactuando con los abonados y con los móviles que les proveen el auxilio mecánico. Por lo tanto, el programa también debe ser interactivo de forma tal que permita ingresar los datos que provienen tanto de las llamadas de los abonados como de las comunicaciones de radio de los móviles.

A grandes rasgos, el programa principal debería verse así:

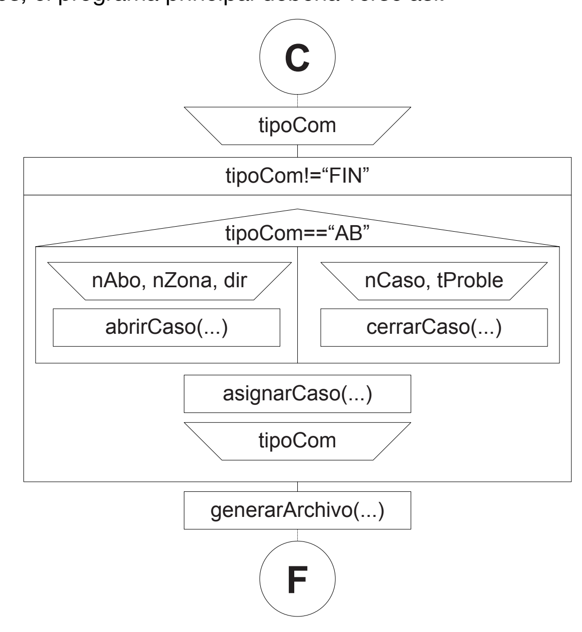


Fig. 11.46 Programa principal sin detalles, estrategia.

El diagrama describe un programa interactivo que espera a que el operador le indique el tipo de comunicación. Cuando llame un abonado se deberá ingresar la cadena “AB”; en cambio, cuando la comunicación provenga desde un móvil se deberá ingresar la cadena “MV”. Luego, según el valor que se ingresó en `tipoCom` (tipo de comunicación), se le pedirá al usuario que ingrese los datos adicionales.

Si llamó un abonado se debe abrir un caso. Si se comunicó un móvil se debe cerrar un caso. Cualquiera de estas acciones probablemente derive en una nueva asignación porque:

- Si se comunicó un móvil significa que quedó libre y, de haber casos en espera en su zona, se le asignará un nuevo caso.
- Si llamó un abonado significa que, en la zona, ahora habrá al menos un caso en espera y si hay algún móvil disponible se le deberá asignar.

Pensemos ahora en una estructura de datos que soporte esta operatoria.

En una zona tenemos abonados que llaman para pedir auxilio y una cantidad finita de móviles que los auxiliarán. Dado que la asignación de casos y la atención a los abonados debe respetar el orden de llegada, tendremos que utilizar dos colas: una cola para encolar los casos abiertos y una cola para encolar los móviles disponibles que se encuentren inactivos. Todo esto multiplicado por 20 ya que ésa es la cantidad de zonas que cubre la empresa.

Lo anterior nos lleva a pensar en dos *arrays* de punteros, de 20 elementos cada uno, de forma tal que cada *array* nos permita implementar 20 colas, una por cada zona.

En el primer *array* (`aCaso[nZona]`) encolaremos los casos a medida que se vayan abriendo. En el segundo (`aMovil[`*numero de zona*`]`) encolaremos los móviles inactivos a medida que estos se vayan comunicando.

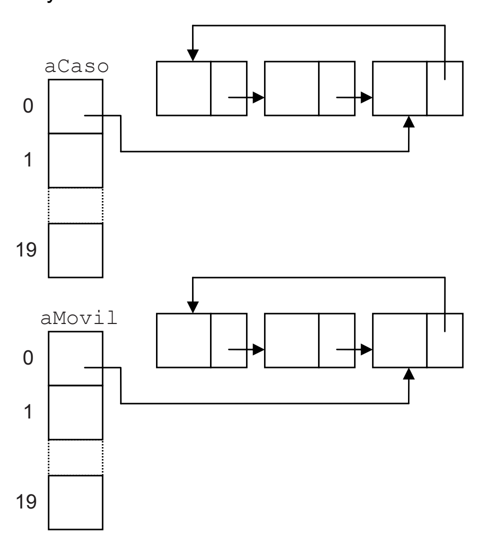


Fig. 11.47 *Arrays* de colas.

Con esta estructura de datos la función `abrirCaso` consistirá en encolar “un caso” en la cola apuntada por `aCaso[nZona]` y la función `cerrarCaso`, entre otras cosas, consistirá en encolar en `aMovil[`*número de zona*`]` el móvil que se acaba de liberar.

En la función `asignarCaso` tenemos que establecer la asignación caso/móvil siempre y cuando en la zona existan casos en espera y móviles disponibles. Esto es fácil de implementar ya que el siguiente `if` nos permitirá determinarlo:

```
if( aCaso[nZona]!=NULL && aMovil[numero de zona]!=NULL )
{
   // hay caso y hay movil para asignar
}
```

Pensemos ahora en el archivo CASOS.dat que nos piden generar grabando un registro por cada uno de los casos atendidos. Para lograrlo tendremos que mantener todos los casos en memoria de forma tal que, al finalizar el programa, estén disponibles para grabarlos en el archivo. Siendo así, cada caso podría registrarse con la siguiente estructura:

```
typedef struct RCaso
{
   long nCaso;     // numero de caso
   long nAbo;      // numero de abonado
   int nZona;      // numero de zona
   int nMovil;     // numero de movil
   int tProblema; // tipo de problema
   int hApert;     // hora de apertura del caso
   int hCierre;    // hora de cierre del caso
}RCaso;
```

Esta estructura, además de coincidir con el tipo de registro del archivo CASOS.dat, nos permitirá llevar el rastro de cada uno de los casos que se vayan abriendo. Claro que para esto, luego de asignarles un móvil, tendremos que agregarlos a una lista de casos.

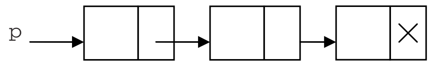


Fig. 11.48 Lista de casos abiertos.

En esta lista, cada nodo tendrá un registro RCaso además de la referencia al siguiente elemento de la lista.

```
typedef struct NCaso
{
   RCaso caso;
   struct NCaso* sig;
}NCaso;
```

La estructura NCaso nos servirá tanto para la lista de casos como para las colas implementadas en el *array* aCaso.

Hagamos un repaso de la estrategia de resolución: cada vez que llame un abonado debemos generar un registro de tipo RCaso y encolarlo en la cola apuntada por aCaso[nZona] que, ahora, tendrá al menos un caso esperando que se le asigne un móvil. Por el momento no contamos con la información suficiente para completar los campos nMovil, tProblema y hCierre del registro así que simplemente debemos omitirlos o asignarles valores absurdos.

Luego, si existe algún móvil disponible en nZona lo asignaremos. Para establecer la asignación móvil/caso desencolaremos un móvil de la cola aMovil[nZona] y desencolaremos un caso de la cola aCaso[nZona]. Ahora podemos completar el campo nMovil del caso que desencolamos y agregarlo a una lista de casos.

Por otro lado, si se recibe la llamada de un móvil será porque se acaba de cerrar un caso. Esto lo reflejaremos buscando (por nCaso) el nodo en la lista de casos y completando el registro con los últimos dos datos que quedaban pendientes: tProblema y hCierre. Para terminar encolaremos el móvil en la cola correspondiente que ahora tendrá, al menos, un móvil disponible. Luego, si en la zona existe algún caso en espera lo desencolaremos para establecer una nueva asignación.

Nos queda un único tema por resolver: las direcciones en donde los móviles deben auxiliar a los abonados. Este dato es fundamental para que el operador, al momento de asignar un móvil, le pueda informar la ubicación exacta en donde el vehículo se encuentra detenido.

Que el dato *dirección exacta* sea irrelevante estadísticamente no significa que lo podamos omitir. Por esto implementaremos un nuevo *array* de colas (`aDir`) donde encolaremos las direcciones. Al separar el dato *dirección exacta* de los demás datos relevantes del caso podremos, por un lado, reutilizar la estructura `NCaso` y, por otro, reducir el uso de memoria.

Al asignar cada caso, desencolaremos la dirección para que el operador la pueda informar y luego la descartaremos.

Pongamos en limpio los tipos de todas las estructuras de datos.

```
// registro del archivo de casos e informacion para los nodos
typedef struct RCaso
{
   long nCaso;    // numero de caso
   long nAbo;     // numero de abonado
   int nZona;     // numero de zona
   int nMovil;    // numero de movil
   int tProblema; // tipo de problema
   int hApert;    // hora de apertura del caso
   int hCierre;   // hora de cierre del caso
}RCaso;

// nodo para la lista y las colas de casos
typedef struct NCaso
{
   RCaso info;
   struct NCaso* sig;
}NCaso;

// nodo para las colas de direcciones
typedef struct NDir
{
   char dir[256];
   struct NDir* sig;
}NDir;

// nodo para las colas de moviles
typedef struct NMovil
{
   int nMovil;
   struct NMovil* sig;
}NMovil;
```

Veamos la declaración de los *arrays* y la lista.

```
#define CANT_ZONAS 20

// array de colas de casos en espera de asignacion
NCaso* aCaso[CANT_ZONAS];
```

```
// array de colas de moviles disponibles
NMov* aMovil[CANT_ZONAS];

// array de colas de direcciones
NDir* aDir[CANT_ZONAS];

// lista de casos asignados
NCaso* p;
```

Ahora sí podemos resolver el problema. Comencemos viendo la versión completa del programa principal.

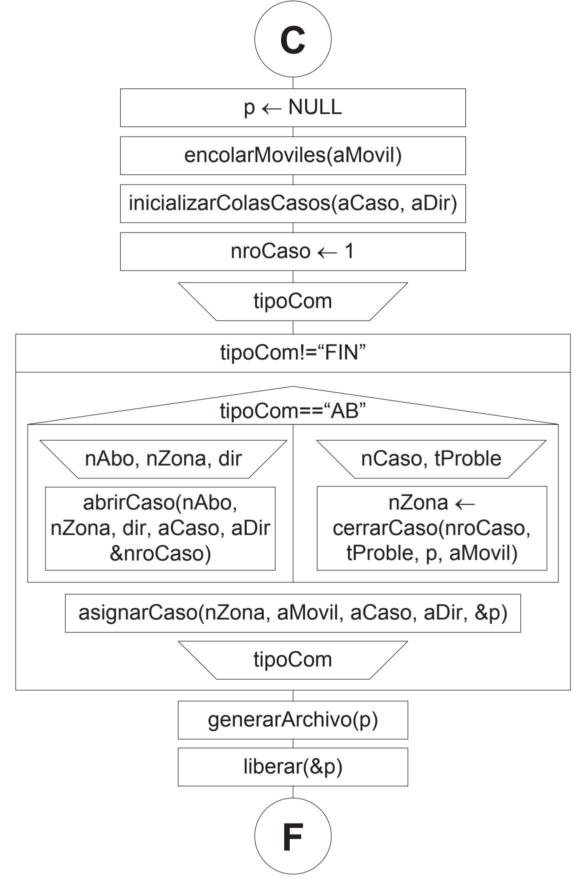


Fig. 11.49 Programa principal.

Primero inicializamos la lista `p` de casos abiertos y/o cerrados. Luego en `encolarMoviles` encolamos en el *array* `aMovil` todos los móviles registrados en el archivo `MOVILES.dat` ya que al comenzar el día todos estarán libres y esperando que el operador les asigne algún caso para auxiliar.

La función `inicializarColasCasos` recorre el *array* `aCaso` asignando `NULL` a cada uno de sus elementos ya que, por el momento, todas las colas estarán vacías.

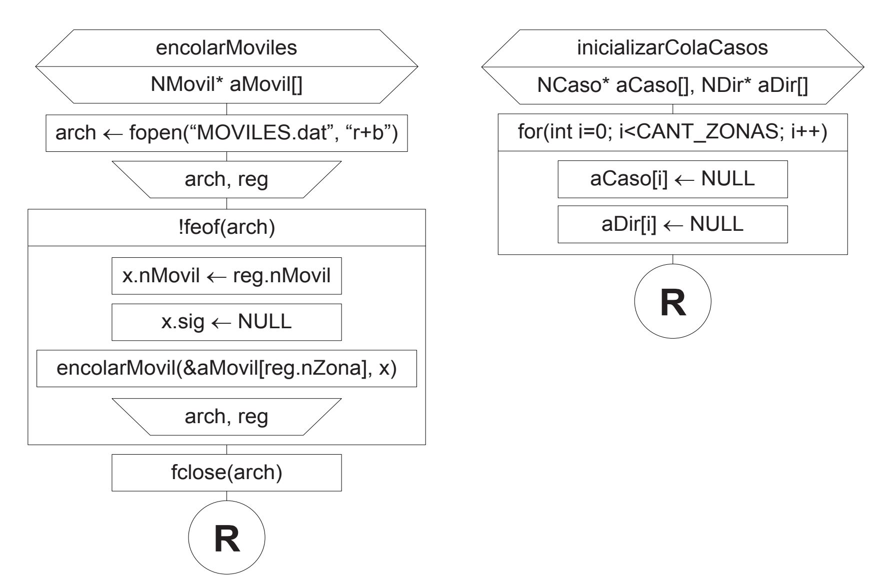


Fig. 11.50 Inicialización de los *arrays* de colas.

El programa principal espera a que el operador ingrese un tipo de comunicación y luego entra en un ciclo que iterará mientras que el valor ingresado sea distinto de "FIN".

Si el valor que ingresó en `tipoCom` es "AB" significa que se comunicó un abonado y debemos abrir un caso. En cambio, si se ingresó un valor diferente a "AB" será porque la comunicación provino desde un móvil, lo que implica cerrar un caso ya abierto.

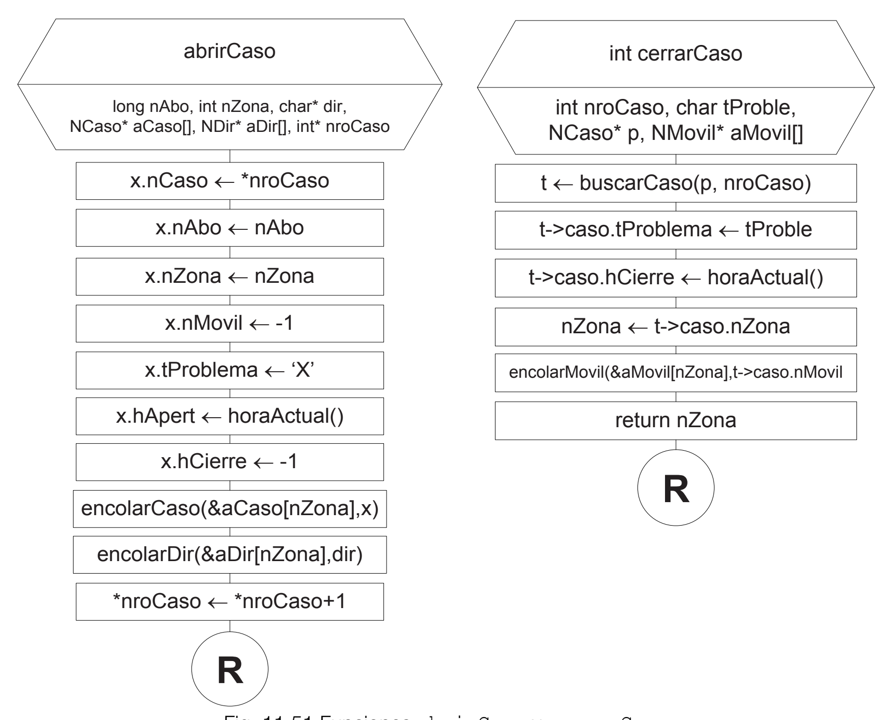


Fig. 11.51 Funciones `abrirCaso` y `cerrarCaso`.

En la función `abrirCaso` armamos un registro de tipo `RCaso` (la variable `x`) y le asignamos valores a todos sus campos excepto a:

- `nMovil` porque el caso aún no tiene asignado un móvil.
- `tProblema` y `hCierre` porque estos datos los tendremos disponibles luego de cerrar el caso.

En la función `cerrarCaso` buscamos el caso en la lista de casos para completar los datos que faltaban: el tipo de problema (`tProblema`) y la hora de cierre (`hCierre`). Luego encolamos el móvil que se acaba de liberar para que espere la asignación de un nuevo caso.

Recordemos que los casos llegan a la lista de casos una vez que se les asignó un móvil. Esto lo hacemos en la función `asignarCaso` cuyo diagrama vemos a continuación.

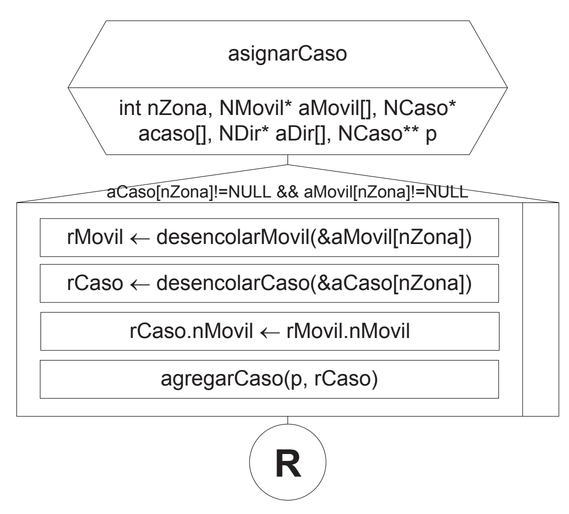


Fig. 11.52 Función `asignarCaso`.

Obviamente, para establecer una asignación móvil/caso debemos tener encolado un caso y un móvil. Si se cumple esto entonces desencolamos un móvil, desencolamos un caso y los vinculamos asignando el número de móvil al campo `nMovil` del registro del caso y agregamos el registro a la lista de casos.

Volviendo al programa principal, solo queda generar el archivo y liberar la memoria dinámica que utiliza la lista de casos. Estas dos funciones se dejan a cargo del lector.

11.10.3 Matriz de pilas

Para terminar veremos un ejercicio cuya solución requerirá usar una matriz de pilas.

Problema 11.6

Para analizar el rendimiento de sus empleados, una empresa decide llevar el registro de las actividades que cada uno desarrolla durante su tiempo de trabajo.

En la empresa trabajan hasta 50 empleados (numerados de 0 a 49) y existen hasta 200 proyectos (numerados de 0 a 199).

Al finalizar la jornada laboral, cada empleado completa uno o varios registros en el archivo `HORAS.dat` cuya estructura es la siguiente:

- Número de empleado.
- Número de proyecto.
- Cantidad de horas.
- Descripción de la tarea realizada.
- Fecha de la tarea.

El archivo se encuentra ordenado naturalmente, es decir, por *fecha* ascendente.

Se pide generar un listado que informe por cada empleado el detalle de las tareas realizadas en cada uno de los proyectos en los que participó. El listado debe estar ordenado por número de empleado, luego por número de proyecto y, finalmente, por *fecha* descendente, es decir, primero las tareas más recientes.

Análisis

La relación “un empleado participa en varios proyectos” y “en un proyecto participan varios empleados” nos lleva a pensar en una matriz. Más aun cuando los identificadores de *empleado* y *proyecto* son valores numéricos y consecutivos.

En una matriz `mat` de 50 filas por 200 columnas cada celda podría representar la relación entre un empleado y un proyecto. Por ejemplo, tomemos un valor entero positivo $i<50$ y otro $j<200$, entonces `mat[i][j]` refleja la participación del empleado *i* en el proyecto *j*. A su vez, el conjunto de celdas que componen la fila *i* representa la participación que el empleado *i* tuvo en cada uno de los diferentes proyectos y el conjunto de celdas que componen la columna *j* representa la participación que ha tenido cada uno de los empleados en el proyecto *j*.

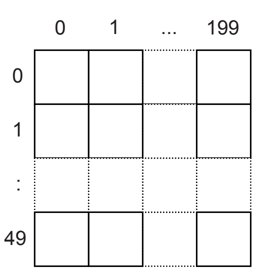


Fig. 11.53 Matriz de relación empleado/proyecto.

Un recorrido por filas y luego por columnas sobre la matriz `mat` nos permitirá avanzar sobre cada uno de los empleados y, por cada uno de estos, evaluar todos los proyectos en los que el empleado participó. Dado que la participación de un empleado en un mismo proyecto puede ser múltiple entonces, en cada celda, tenemos que guardar el conjunto de tareas que el empleado haya realizado en el proyecto. Ver Fig.11.54.

En la figura vemos que el empleado número 0 (fila 0) participó dos veces en el proyecto número 1 (columna 1) y participó 3 veces en el proyecto número 199. En cambio, no participó en el proyecto número 0.

Esta estructura de datos será el soporte del algoritmo que utilizaremos para procesar el archivo `HORAS.dat` y emitir el listado que nos piden en el enunciado.

La estrategia será la siguiente: vamos a recorrer secuencialmente el archivo `HORAS.dat`. Dado que cada registro representa una tarea desarrollada por un empleado en un proyecto utilizaremos el número de empleado y el número de proyecto como índices de acceso directo a la matriz. Luego, en la lista apuntada `mat[i][j]` , siendo *i* el número de empleado y *j* el número de proyecto, agregaremos un nodo para representar una tarea del empleado en el proyecto.

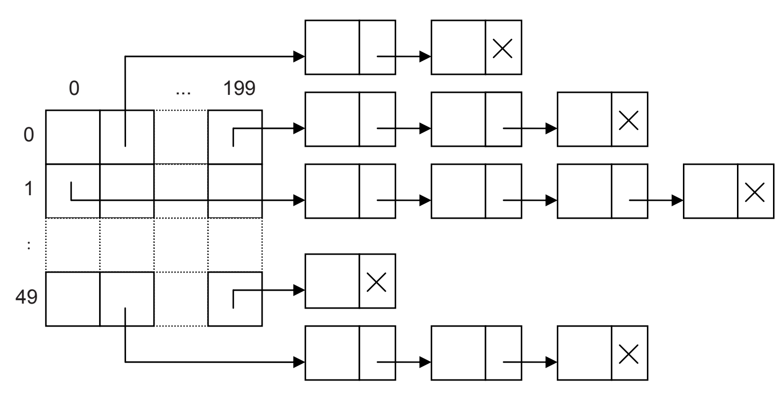


Fig. 11.54 Matriz de pilas.

Así, al finalizar el proceso de `HORAS.dat` tendremos una matriz en la que de cada celda se desprende una lista que representa la colección de tareas que un empleado realizó en un proyecto.

Como el archivo está ordenado por "fecha ascendente", los nodos que agreguemos en `mat[`*i*`][`*j*`]` también lo estarán. Sin embargo, si en lugar de apuntar a listas consideramos que las celdas apuntan a pilas entonces al desapilar cada nodo lo obtendremos en orden inverso, esto es: por "fecha descendente", tal como se solicita en el enunciado.

Veamos la definición de tipos de datos: la estructura del archivo y la estructura de la matriz.

```
// constantes
#define MAX_EMPLEADOS  50
#define MAX_PROYECTOS 200

// estructura de los registros del archivo HORAS.dat
typedef struct RHora
{
   int nroEmple;
   int nroProye;
   int cantHs;
   char descrip[256];
   long fecha;
}RHora;

// tipo de datos de la matriz
typedef struct NMat
{
   int pos;
   struct NMat* sig;
}NMat;
```

La declaración de la matriz será la siguiente:

```
NMat* mat[MAX_EMPLEADOS][MAX_PROYECTOS];
```

Para reducir el uso de memoria solo apilaremos las posiciones de los registros. La matriz, entonces, será "una matriz de pilas de posiciones de registros" lo que nos obligará a recorrer dos veces el archivo: Durante la primera pasada cargaremos la matriz y durante la segunda, por cada posición que desapilemos, haremos un acceso directo al archivo para obtener los datos completos del registro y mostrarlos por pantalla.

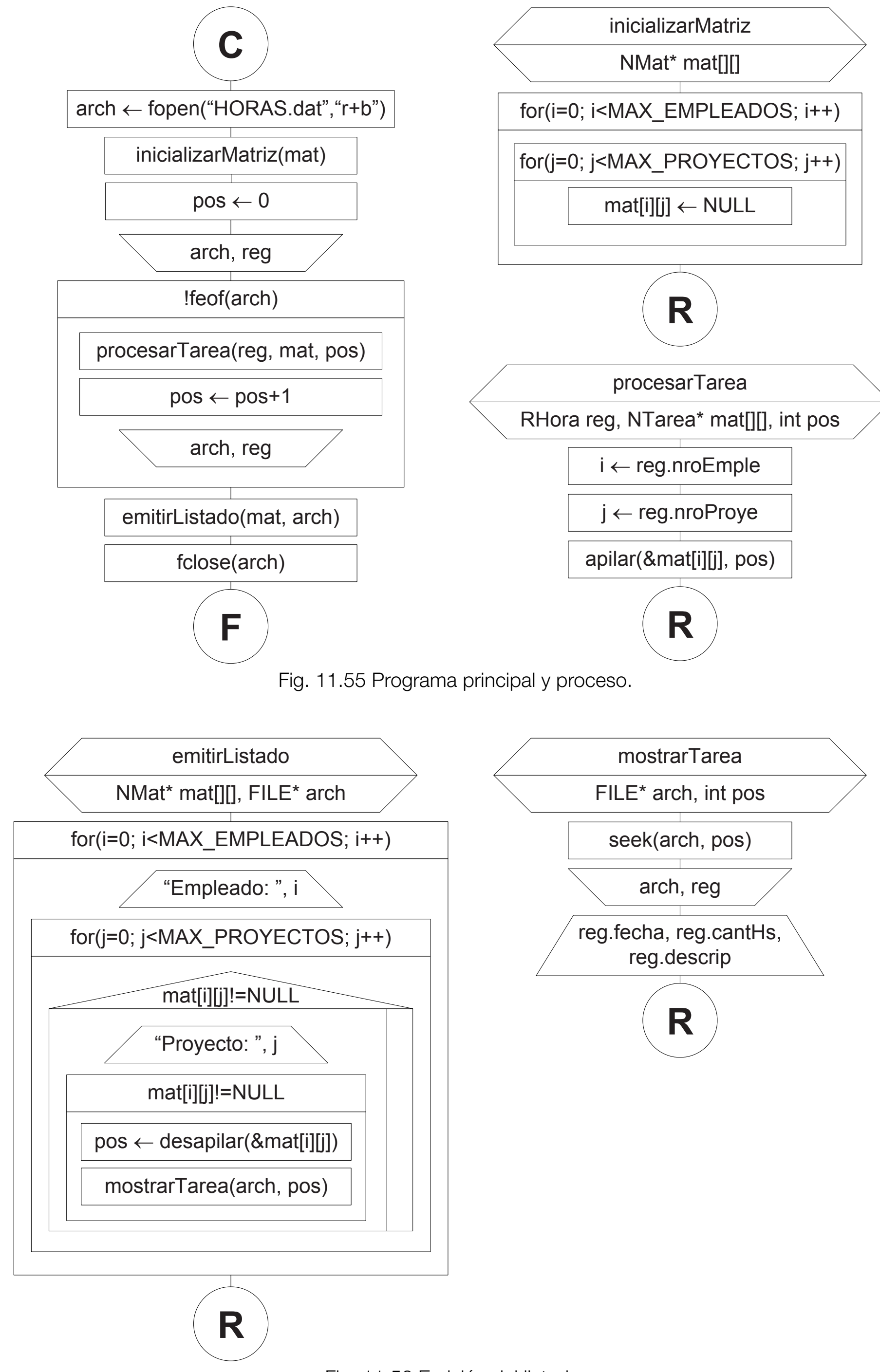


Fig. 11.55 Programa principal y proceso.

Fig. 11.56 Emisión del listado.

11.11 Resumen

En este capítulo estudiamos estructuras dinámicas que nos permitieron guardar en memoria colecciones de datos de tamaño indefinido. Como todas se forman concatenando nodos resulta que, en definitiva, una estructura de datos dinámica no es más que un puntero a un nodo más un conjunto de operaciones asociadas.

En los próximos capítulos, profundizaremos los conceptos de encapsulamiento que estudiamos en el capítulo de “Tipo Abstracto de Dato”. Veremos elementos de programación orientada a objetos y comenzaremos a programar en Java, previo paso por C++.

11.12 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

11.12.1 Mapa conceptual

11.12.2 Autoevaluaciones

11.12.3 Presentaciones*

12

Encapsulamiento a través de clases y objetos

Contenido

Objetivos del capítulo

- Conocer la estructura de las clases.
- Encapsular lógica y complejidad mediante el uso clases y objetos.
- Determinar la necesidad y disponer de tipos de datos genéricos.
- Introducir al alumno en el lenguaje de programación Java, previo paso por C++.

Competencias específicas

- Comprender, describir y modelar los conceptos principales del paradigma de programación orientado a objetos y aplicarlos a situaciones de la vida real.

12.1 Introducción

La programación estructurada, tal como la estudiamos hasta aquí, hace énfasis en los procesos. Los módulos o procesos, que se implementan como funciones en C, reciben sus parámetros y realizan la tarea para la cual fueron diseñados en forma adecuada.

Lo anterior suena razonable hasta que nos comenzamos a formular preguntas como esta: ¿Por qué si quiero usar una cadena de caracteres, tengo que preocuparme por agregarle el '\0' al final, asignar memoria para que la contenga y conocer funciones de biblioteca tales como `strcpy`, `strcat`, `strlen`, etcétera?

Justamente, las clases permiten encapsular toda esa complejidad, propia de la implementación, pero totalmente ajena para el programador.

En este capítulo utilizaremos C++ y Java para estudiar la primera de las premisas de la teoría de objetos: el encapsulamiento.

Comenzaremos codificando en C++, pero con un estilo de programación muy particular, casi idéntico al estilo que se utiliza para programar en Java, ya que aquí haremos la transición entre ambos lenguajes.

Cabe aclarar también que los temas que vamos a exponer se tratarán con cierta ligereza. Esto nos permitirá concentrarnos en los conceptos y tener una primera aproximación a la programación orientada a objetos, tema que analizaremos en detalle en el Capítulo 14.

12.2 Clases y objetos



Una clase es una especie de estructura que agrupa datos y funciones. Un objeto es una variable cuyo tipo de datos es una clase.

Cuando hablamos de programación orientada a objetos, hablamos de clases y objetos. Para hacerlo rápido y no dar demasiadas vueltas sobre el tema diremos lo siguiente:

Una clase es un tipo de datos definido por el programador, algo así como un `struct` de C. Por lo tanto, llamamos objeto a las variables cuyo tipo de dato es una clase.

12.2.1 Las clases

Una clase es un tipo de datos definido por el programador. Una especie de estructura en la que, además de definir campos, definimos las funciones a través de las cuales permitiremos manipular los valores de esos campos.

En lugar de hablar de "campos" hablaremos de "variables de instancia" y en lugar de hablar de "funciones" hablaremos de "métodos". Esto solo es una cuestión semántica ya que funciones y métodos, variables de instancia y campos son exactamente lo mismo.

Para entender de qué estamos hablando, veamos el siguiente programa en donde utilizamos la clase `Cadena` (que analizaremos y desarrollaremos más abajo) para concatenar varias cadenas de caracteres.

```
#include <stdio.h>
#include "Cadena.h"

int main()
{
   // definimos un objeto de tipo Cadena inicializado con "Hola,"
   Cadena* s = new Cadena("Hola,");

   // le concatenamos otra cadena
   s->concatenar(" que tal?");
```

```
    // le concatenamos otra cadena
    s->concatenar(" Todo bien?");

    // le concatenamos otra cadena
    s->concatenar(" Me alegro mucho!");

    // mostramos su contenido
    printf("%s\n", s->toString());

    // eliminamos el objeto, ya no lo necesitamos
    delete s;

    return 0;
}
```

Como `Cadena` es una clase, entonces `s` (variable de tipo `Cadena`) es un objeto al que le asignamos un valor inicial y luego le concatenamos varias cadenas más. Al final del programa mostramos el resultado en la consola.

Notemos que en ningún momento nos preocupamos por declarar una variable de tipo `char*` ni por asignar un '\0' ni mucho menos por hacer `malloc`. Simplemente nos limitamos a invocar una serie de métodos sobre el objeto `s`.

Evidentemente, toda la complejidad que en C implica inicializar y concatenar cadenas de caracteres está encapsulada dentro de los métodos de la clase `Cadena` cuyo código analizaremos enseguida.

12.2.2 Miembros de la clase

Como comentamos más arriba, una clase se compone de variables de instancia (o campos) y métodos (o funciones). Al conjunto compuesto por las variables de instancia y los métodos de una clase lo llamamos "miembros de la clase".



Al conjunto compuesto por las variables de instancia y los métodos de una clase lo llamamos "miembros de la clase".

Veamos la estructura de la clase `Cadena`, que se compone de los siguientes miembros:

Variables miembro de `Cadena` = { `cad` }

Métodos miembro de `Cadena` = { `Cadena`, `~Cadena`, `concatenar`, `toString` }

```
class Cadena
{
    // variable de instancia
    private: char* cad;

    // constructor
    public: Cadena(const char* cadInicial)
    {
        // :
        // aqui va el codigo del constructor
        // :
    }

    // destructor
    public: ~Cadena()
    {
        // :
        // aqui va el codigo del destructor
        // :
    }
```

```
    // metodo concatenar
    public: void concatenar(const char* cadConcat)
    {
        // :
        // aqui va el codigo del metodo concatenar
        // :
    }

    // metodo toString
    public: char* toString()
    {
        // :
        // aqui va el codigo del metodo toString
        // :
    }
};
```

El código anterior solo refleja la estructura de la clase `Cadena`. Falta la codificación de todos los métodos que, obviamente, analizaremos más adelante.

Sin embargo, esta estructura nos permite apreciar lo siguiente:

- La variable de instancia `cad` está definida como `private`.
- Los métodos están definidos como `public`.



Las variables de instancia se suelen definir como miembros privados para limitar su accesibilidad y así evitar que puedan ser manipuladas desde afuera de la clase. Los miembros públicos están expuestos y constituyen la interfaz de los objetos de la clase.

Como explicamos más arriba, los métodos deben ser los únicos responsables de manipular los valores de los campos. Las variables de instancia se suelen definir como miembros privados (`private`) para limitar su accesibilidad y así evitar que puedan ser manipuladas desde afuera de la clase como, por ejemplo, desde el programa principal.

Por el contrario, los métodos se definen como públicos (`public`), lo que nos permite aplicarlos sobre los objetos de la clase y así interactuar.

Decimos entonces que los miembros privados están encapsulados y los miembros públicos están expuestos y constituyen lo que llamaremos la interfaz de los objetos de la clase.

12.2.3 Interfaz y encapsulamiento

Para comprender mejor los conceptos de interfaz y encapsulamiento, podemos pensar, por ejemplo, en un teléfono celular. El teléfono tiene los botones numéricos a través de los que podemos marcar un número y luego dos botones: uno verde para establecer la llamada y uno rojo para cortar la comunicación.

Con esto, a cualquier usuario común le resulta muy fácil manejar un teléfono celular ya que, usando adecuadamente este conjunto de botones, puede marcar un número y luego establecer la comunicación deseada.

Obviamente, nadie, salvo un estudiante de ingeniería en comunicaciones, se preocupará por entender el proceso que se origina dentro del teléfono luego de que presionamos el botón verde para establecer la llamada. Simplemente, nos abstraemos del tema ya que ese no es nuestro problema.

Decimos entonces que el conjunto de botones numéricos más los botones rojo y verde constituyen la interfaz del objeto teléfono celular. El usuario (nosotros) interactúa con el objeto a través de su interfaz y no debe abrir la carcasa del teléfono para ver ni mucho menos para tocar su implementación (cables, chips, transistores, etc.).

Justamente, en el teléfono celular, los botones están expuestos y los componentes electrónicos son internos y están protegidos (encapsulados) dentro de la carcasa para que nadie los pueda tocar.

Ahora sí podremos ver el código fuente completo de la clase `Cadena` donde podremos observar que dentro de los métodos accedemos a la variable de instancia `cad` como si esta fuese global.

```
#include <stdio.h>
#include <string.h>

class Cadena
{
   // variables de instancia
   private: char* cad;

   // constructor
   public: Cadena(const char* cadInicial)
   {
      cad=(char*)malloc(sizeof(char)*strlen(cadInicial)+1);
      strcpy(cad,cadInicial);
   }

   // destructor
   public: ~Cadena()
   {
      free(cad);
   }

   // metodos...
   public: Cadena* concatenar(const char* cadConcat)
   {
      int size=strlen(cad)+strlen(cadConcat)+1;
      char* aux=(char*)malloc(sizeof(char)*size);
      strcpy(aux,cad);
      strcat(aux,cadConcat);
      cad=aux;

      return this;
   }

   public: char* toString()
   {
      return cad;
   }
};
```

12.2.4 Estructura de una clase

Analicemos por partes el código de la clase `Cadena` que, siguiendo los lineamientos de este estilo particular de codificación C++, debe estar ubicada en el archivo `Cadena.h`.

Comenzamos definiendo el nombre de la clase y las variables de instancia.

```
#include <stdio.h>
#include <string.h>

class Cadena
{
   // variables de instancia
   private: char* cad;
```

Como se comentó más arriba, los campos (o variables de instancia) son privados ya que no deben ser accedidos desde afuera de la clase. Por este motivo, anteponemos la palabra `private` a la declaración de la variable `cad`. En cambio, los métodos son públicos porque constituyen la interfaz a través de la cual el programador que use la clase podrá interactuar con los objetos.

12.2.5 El constructor y el destructor

Los objetos se construyen, se usan y luego se destruyen. Para construir un objeto, utilizamos el operador `new` mientras que para destruirlo utilizamos el operador `delete`. Estos operadores reciben como parámetro a los métodos "constructor" y "destructor" respectivamente.

El constructor es un método especial a través del cual podemos asignar valores iniciales a las variables de instancia del objeto. Análogamente, el destructor también es un método especial donde podremos liberar los recursos que el objeto haya obtenido.

```
   // constructor
   public: Cadena(const char* cadInicial)
   {
      cad=(char*)malloc(sizeof(char)*strlen(cadInicial)+1);
      strcpy(cad,cadInicial);
   }

   // destructor
   public: ~Cadena()
   {
      free(cad);
   }
```

Decimos que el constructor es un método especial con las siguientes características:

1. Solo puede invocarse como argumento del operador `new`.
2. El nombre del constructor debe coincidir con el nombre de la clase.
3. El constructor no tiene valor de retorno.

Decimos que el destructor es un método especial con las siguientes características:

1. Solo puede invocarse como argumento del operador `delete`.
2. El nombre del destructor debe comenzar con el carácter "~" (léase carácter "tilde" o "ñuflo") seguido del nombre de la clase.
3. El destructor no tiene valor de retorno.

En el código del constructor, vemos que recibimos un `char*`, que será el valor inicial que tomará la variable de instancia `cad`. Claro que para asignarlo tenemos que dimensionar la cadena con `malloc` y luego copiar carácter a carácter con `strcpy` que, además, asignará un '\0' al final.

Observemos que toda esta complejidad resulta totalmente transparente para el programa principal, donde simplemente definimos el objeto `s` de la siguiente manera:

```
Cadena* s = new Cadena("Hola,");
```

Cuando ya no necesitemos usar la cadena `s`, tenemos que destruirla. Esto lo haremos invocando al operador `delete` pasándole como argumento el objeto que queremos destruir. El operador `delete` invocará al destructor del objeto donde, volviendo al código, vemos que usamos la función `free` para liberar el espacio de memoria direccionado por `cad`.

12.2.6 Los métodos

Los métodos, incluidos el constructor y el destructor, son los únicos responsables de manipular los valores de las variables de instancia.

Notemos que dentro de los métodos tenemos acceso a las variables de instancia como si estas fuesen variables globales.

```
    // metodos...
    public: void concatenar(const char* cadConcat)
    {
        int size=strlen(cad)+strlen(cadConcat)+1;
        char* aux=(char*)malloc(sizeof(char)*size);
        strcpy(aux,cad);
        strcat(aux,cadConcat);
        cad=aux;
    }

    public: char* toString()
    {
        return cad;
    }
};
```

Observemos toda la complejidad que encapsula el método `concatenar`, donde redimensionamos el espacio de memoria direccionado por `cad` para que pueda contener, además, la cadena que se recibe como parámetro.

Gracias a este método, concatenar una cadena en el programa principal resulta ser algo trivial como esto:

```
s->concatenar(" que tal?");
```

Por último, para tener acceso al valor de la variable `cad` y usarla, por ejemplo, en un `printf`, el método `toString` retorna su valor, que es de tipo `char*`.

```
printf("%s\n", s->toString());
```

12.2.7 Los objetos

Como dijimos más arriba, un objeto es una variable cuyo tipo de dato es una clase. La clase define las variables de instancia y los métodos necesarios para manipular sus valores.

Veamos el siguiente código:

```
Cadena* s1 = new Cadena("Pablo");
Cadena* s2 = new Cadena("Juan");
```

Aquí tenemos dos objetos tipo `Cadena`: `s1` y `s2`, cada uno de los cuales tiene su propio valor en la variable `cad`. Es decir: la variable `cad` de `s1` tiene la dirección de la cadena "Pablo" mientras que la variable `cad` de `s2` tiene la dirección de la cadena "Juan".

Decimos entonces que `s1` y `s2` son instancias de la clase `Cadena` ya que cada una mantiene valores propios e independientes en sus variables de instancia.

Por supuesto, también es correcto referirse a `s1` y `s2` como objetos de tipo `Cadena`.

12.2.8 Instanciar objetos

Si bien los términos "objeto" e "instancia" se usan como sinónimos, existe una sutil diferencia entre ambos: un objeto puede no estar instanciado y, por otro lado, una instancia puede no estar siendo referenciada por un objeto.

Veamos: en la siguiente línea de código declaramos un objeto `s` de tipo `Cadena`, pero lo dejamos sin instanciar (no invocamos a su constructor, no invocamos al operador `new`).

```
// declaramos el objeto s sin instanciarlo
Cadena* s;
```

En la siguiente línea de código instanciamos al objeto `s` declarado más arriba.

```
// instanciamos el objeto
s = new Cadena("Hola");
```

Más adelante, veremos casos en los que instanciamos la clase sin tener objetos que apunten a esas instancias.

12.2.9 Operadores `new`, `delete` y punteros a objetos

El operador `new` permite instanciar objetos. Cuando hacemos:

```
Cadena* s = new Cadena("Hola");
```

usamos el operador `new` para asignar memoria dinámicamente y asignamos a `s` la dirección del espacio de memoria recientemente asignada. La variable `s` es en realidad un puntero, por lo que el operador `new` de C++ es comparable a la función `malloc` de C.

Por otro lado, el operador `delete`, luego de invocar al destructor del objeto, libera la memora direccionada por este.

12.2.10 Sobrecarga de métodos

Sobrecargar un método implica escribirlo dos o más veces dentro de la misma clase con diferentes cantidades y/o tipos de parámetros y diferentes implementaciones.

Por ejemplo, podemos sobrecargar el método `concatenar` en la clase `Cadena` para que, además, permita concatenar valores enteros.

```
   // permite concatenar una cadena
   public: void concatenar(const char* cadConcat)
   {
      int size=strlen(cad)+strlen(cadConcat)+1;
      char* aux=(char*)malloc(sizeof(char)*size);
      strcpy(aux,cad);
      strcat(aux,cadConcat);
      cad=aux;
   }

   // permite concatenar un entero
   public: void concatenar(int n)
   {
      // convertimos el entero a cadena
      char sNum[10];
      itoa(n,sNum,10);

      // invocamos a la otra version de concatenar pasandole
      // la cadena que representa al entero
      concatenar(sNum);
   }
```

Ahora, en un programa podemos concatenar tanto cadenas como números enteros.

```
Cadena* s = new Cadena("Estamos en diciembre de ");
s->concatenar(2011);
```

12.3 Encapsulamiento de estructuras lineales

Las clases son una gran herramienta para encapsular la complejidad de las estructuras de datos que estudiamos anteriormente: pilas, colas y listas.

Aquí analizaremos cómo encapsular la estructura lineal más simple de todas: la pila que, como ya sabemos, se implementa sobre una lista enlazada de nodos y define dos operaciones: `poner` y `sacar`.

12.3.1 Análisis de la clase `Pila`

Analicemos entonces la clase `Pila` que debe estar contenida en el archivo `Pila.h` cuya estructura, a grandes rasgos, será la siguiente:

```
class Pila
{
   private Nodo* p;

   public: void poner(int v)
   {
      // aqui va la implementacion del metodo
   }

   public: int sacar()
   {
      // aqui va la implementacion del metodo
   }
}
```

Contando con la clase `Pila`, utilizar una pila de enteros en nuestro programa resultará extremadamente simple:

```
#include "Pila.h"

int main()
{
   Pila* pila = new Pila();
   pila->poner(3);
   pila->poner(2);
   pila->poner(1);

   printf("%d\n", pila->sacar());
   printf("%d\n", pila->sacar());
   printf("%d\n", pila->sacar());

   return 0;
}
```

Veamos la implementación de la clase.

```
#include <stdio.h>
#include <stdlib.h>

// estructura Nodo
typedef struct Nodo
{
   int valor;
   struct Nodo* sig;
}Nodo;

// clase Pila
class Pila
{
   private: Nodo* p;

   public: Pila()
   {
      p = NULL;
   }

   public: void poner(int v)
   {
      Nodo* nuevo = (Nodo*) malloc(sizeof(Nodo));
      nuevo->valor = v;
      nuevo->sig = p;
      p = nuevo;
   }

   public: int sacar()
   {
      Nodo* aux = p;
      int ret = aux->valor;
      p = aux->sig;
      free(aux);
      return ret;
   }
};
```

Observemos que el manejo de punteros dentro de los métodos `poner` y `sacar` es mucho más simple que cuando implementamos estas operaciones como funciones sueltas. Esto se debe a que `p` (puntero al primer nodo de la pila) ahora es una variable de instancia a la que los métodos pueden acceder directamente para modificar su valor. Es decir, no tenemos que pasar punteros por referencia.

12.3.2 Templates y generalizaciones

Hasta aquí hemos venido arrastrando un problema que, aun sin haberlo mencionado como tal, más de un lector habrá detectado.

La clase `Pila`, por ejemplo, nos permite apilar elementos de tipo `int`, pero no funciona con valores de otro tipo de datos. Esto se debe a que el tipo de datos del campo `valor` de la estructura `Nodo` es `int`. Está *hardcodeado*.

```
typedef struct Nodo
{
   int valor;
   struct Nodo* sig;
}Nodo;
```

Si pudiésemos hacer que el tipo de datos del campo `valor` fuera variable, entonces la clase `Pila` servirá también para apilar datos de diferentes tipos siempre y cuando especifiquemos el tipo de datos concreto con el que queremos trabajar.

Cuando los tipos de datos con los que trabaja una clase son variables y pueden especificarse como argumentos, decimos que la clase es un *template*.



Cuando los tipos de datos con los que trabaja una clase son variables y pueden especificarse como argumentos, decimos que la clase es un *template*.

Veamos primero cómo quedaría un programa que utilice instancias de una clase `Pila` genérica (un *template*) para apilar enteros, *doubles* y cadenas.

```
#include <stdlib.h>
#include <stdio.h>
#include "Pila.h"
#include "Cadena.h"

int main()
{
   // una pila de enteros
   Pila<int>* pila1 = new Pila<int>();
   pila1->poner(3);
   pila1->poner(2);
   pila1->poner(1);

   printf("%d\n", pila1->sacar());
   printf("%d\n", pila1->sacar());
   printf("%d\n", pila1->sacar());

   // una pila de doubles
   Pila<double>* pila2 = new Pila<double>();
   pila2->poner(23.1);
   pila2->poner(22.2);
   pila2->poner(21.3);

   printf("%lf\n", pila2->sacar());
   printf("%lf\n", pila2->sacar());
   printf("%lf\n", pila2->sacar());
```

```
   // una pila de cadenas
   Pila<Cadena*>* pila3 = new Pila<Cadena*>();
   pila3->poner(new Cadena("tres"));
   pila3->poner(new Cadena("dos"));
   pila3->poner(new Cadena("uno"));

   printf("%s\n", (pila3->sacar())->toString());
   printf("%s\n", (pila3->sacar())->toString());
   printf("%s\n", (pila3->sacar())->toString());
}
```

La salida de este programa será la siguiente.

```
1
2
3
21.30
22.20
23.10
uno
dos
tres
```

Como vemos, al momento de declarar el objeto de tipo `Pila` debemos especificar el tipo de datos de los elementos que vamos a apilar.

```
// declaramos una pila de enteros
Pila<int>* pila1;

// declaramos una pila de doubles
Pila<double>* pila2;

// declaramos una pila de cadenas
Pila<Cadena*>* pila3;
```

Los *templates* permiten *parametrizar* para los tipos de datos de los miembros de las clases. Podemos hacer que una clase sea un *template*, pero no podemos hacer esto con un `struct`.

Como en nuestro caso tenemos que *parametrizar* el tipo de datos del campo `valor` de la estructura `Nodo`, la convertiremos en un *template* codificándola en el archivo `Nodo.h` como una clase con sus variables de instancia públicas. Con esto, obtendremos una estructura idéntica a `struct Nodo`, pero con el tipo de datos del campo `valor` parametrizado.

```
template <typename T>
class Nodo
{
   public: T valor;
   public: Nodo<T>* sig;
};
```

Ahora cada vez que definamos una variable de tipo Nodo*, tendremos que pasarle como parámetro el tipo de datos que tomará el campo `valor`.

Por ejemplo:

```
Nodo<int>* aux;
```

Luego de esta declaración, `aux->valor` es de tipo `int`.
En cambio, si declaramos la variable `aux` de la siguiente manera:

```
Nodo<double>* aux;
```

entonces `aux->valor` será de tipo `double`.

Ahora podemos replantear la clase `Pila` como un *template* de la siguiente manera:

```
#include "Nodo.h"

template <typename T>
class Pila
{
   private: Nodo<T>* p;

   public: Pila()
   {
      p = NULL;
   }

   // el tipo de dato de v es T
   public: void poner(T v)
   {
      // como Nodo ahora es una clase podemos instanciarlo con new
      Nodo<T>* nuevo = new Nodo<T>();
      nuevo->valor = v;
      nuevo->sig = p;
      p = nuevo;
   }

   // el tipo de dato del valor de retorno del metodo ahora es T
   public: T sacar()
   {
      Nodo<T>* aux = p;
      T ret = aux->valor;

      p = aux->sig;
      free(aux);

      return ret;
   }
};
```

Analicemos paso a paso el nuevo código de la clase `Pila`.

```
#include "Nodo.h"

template <typename T>
class Pila
{
   private: Nodo<T>* p;

   public: Pila()
   {
      p = NULL;
   }
```

La clase `Pila` ahora es un *template*; por lo tanto, cuando declaremos objetos tendremos que especificar un valor concreto para el parámetro `T`. Dentro de la clase, `T` es un tipo de datos variable.

El método `poner` recibe un valor `v` de tipo `T`.

```
// el tipo de dato de v es T
public: void poner(T v)
{
   Nodo<T>* nuevo = new Nodo<T>();
   nuevo->valor = v;
   nuevo->sig = p;
   p = nuevo;
}
```

y el método `sacar` retorna un valor de tipo `T`.

```
   // el tipo de dato del valor de retorno del metodo ahora es T
   public: T sacar()
   {
      Nodo<T>* aux = p;
      T ret = aux->valor;

      p = aux->sig;
      free(aux);

      return ret;
   }
};
```

12.4 El lenguaje de programación Java

Java es un lenguaje de programación de propósitos generales cuya sintaxis es casi idéntica a la de C++.

El hecho de haber estudiado C y luego, en este mismo capítulo, haber visto C++ codificado con "estilo Java" nos permitirá programar en Java tan solo mencionando dos diferencias que existen entre estos lenguajes.

Diferencia 1:

En C++ declaramos los objetos explícitamente como punteros. Cuando hacemos:

```
Nodo* aux = new Nodo();
```

estamos diciendo que `aux` es un puntero a un objeto de tipo `Nodo`, luego asignamos memoria dinámicamente y colocamos en `aux` su dirección.

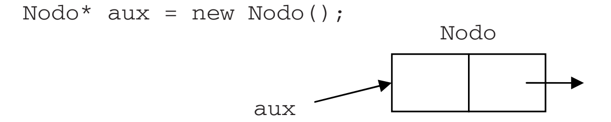


Fig. 12.1 Puntero a un objeto `Nodo` en C++.

En Java no existe la posibilidad de que un objeto no sea un puntero. Todos los objetos son punteros; por lo tanto, no se utiliza el asterisco para declararlos.



En Java todos los objetos son punteros; por lo tanto, no se utiliza el asterisco para declararlos.

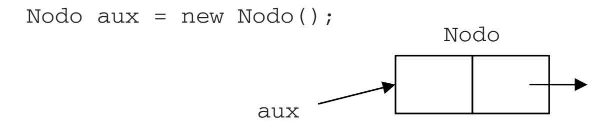


Fig. 12.2 Puntero a un objeto `Nodo` en Java.

Diferencia 2:

En Java la responsabilidad de liberar la memoria que ya no usamos no es nuestra; el mismo lenguaje se ocupa de liberar por nosotros aquella memoria que vamos dejando sin referencia. En consecuencia, no existe nada parecido a la función `free` de C o al operador `delete` de C++. Tampoco existe el concepto de "método destructor", aunque en el próximo capítulo veremos que, de alguna manera, el método `finalize` es comparable a la idea de "método destructor" de C++.

Una vez marcadas estas diferencias, podemos codificar directamente la clase `Pila` comenzando por su versión más simple, que solo permite apilar valores enteros.

La clase `Nodo`:

```
public class Nodo
{
   public int valor;
   public Nodo sig;
}
```

La clase `Pila`:

```
public class Pila
{
   private Nodo p;

   public Pila()
   {
      p = null; // null es en minuscula
   }

   public void poner(int v)
   {
      Nodo nuevo = new Nodo();
      nuevo.valor = v;
      nuevo.sig = p;
      p = nuevo;
   }

   public int sacar()
   {
      Nodo aux = p;
      int ret = aux.valor;
      p = p.sig;
      return ret;
   }
}
```

12.4.1 El programa principal en Java

La estructura del lenguaje Java es más rígida que la de C++. De hecho, se dice que C++ es un superconjunto de C, un lenguaje híbrido que soporta objetos. En C++ podemos compilar código C, es decir, un programa estructurado.

En Java solo programamos clases, no funciones sueltas. Por esto, el programa principal se desarrolla dentro del método `main` de una clase.

```
public class TestPila
{
   public static void main(String[] args)
   {
      Pila p = new Pila();
      p.poner(3);
      p.poner(2);
      p.poner(1);

      System.out.println( p.sacar() );
      System.out.println( p.sacar() );
      System.out.println( p.sacar() );
   }
}
```

Aprovecharemos este programa para marcar otras diferencias:

En Java existe la clase `String` que facilita el manejo de cadenas de caracteres a un alto nivel. El método `main` siempre recibe un `String[]` que nos permitirá acceder a los argumentos pasados en línea de comandos.

El "printf" de Java es `System.out.println`. Podemos usarlo, por ejemplo, así:

```
String nombre = "Pablo";
int edad = 40;
System.out.println("Yo soy "+nombre+" y tengo "+edad+" anios");
```

Para los más nostálgicos existe el método `System.out.printf` que imita al viejo y nunca bien ponderado `printf` de C.

```
String nombre = "Pablo";
int edad = 40;

System.out.printf("Yo soy %s y tengo %d anios\n",nombre, edad);
```

12.4.2 Templates en C++, generics en Java

En Java también podemos *parametrizar* los tipos de datos de los miembros de las clases haciendo que las clases sean genéricas. El concepto es, exactamente, el mismo que el de usar *templates* en C++ y la sintaxis, prácticamente, también lo es.

Veamos entonces la segunda versión de la clase `Pila`, ahora genérica.

Primero la clase `Nodo`, genérica en `T`.

```
public class Nodo<T>
{
   public T valor;
   public Nodo<T> sig;
}
```

Ahora la clase `Pila`.

```
public class Pila<T>
{
   private Nodo<T> p;

   public Pila()
   {
      p = null;
   }

   public void poner(T v)
   {
      Nodo<T> nuevo = new Nodo<T>();
      nuevo.valor = v;
      nuevo.sig = p;
      p = nuevo;
   }

   public T sacar()
   {
      Nodo<T> aux = p;
      T ret = aux.valor;
      p = p.sig;
      return ret;
   }
}
```

Como vemos, ni siquiera es necesario escribir la palabra *template*. Alcanza con especificar el nombre del parámetro entre los símbolos “menor” y “mayor”.

Por último, veremos un programa donde utilizamos una pila de enteros, una pila de *doubles* y una pila de cadenas de caracteres.

```
public class TestPila
{
   public static void main(String[] args)
   {
      // una pila de enteros
      Pila<Integer> p1 = new Pila<Integer>();
      p1.poner(3);
      p1.poner(2);
      p1.poner(1);

      System.out.println( p1.sacar() );
      System.out.println( p1.sacar() );
      System.out.println( p1.sacar() );

      // una pila de doubles
      Pila<Double> p2 = new Pila<Double>();
      p2.poner(33.1);
      p2.poner(32.2);
      p2.poner(31.3);
```

```
        System.out.println( p2.sacar() );
        System.out.println( p2.sacar() );
        System.out.println( p2.sacar() );

        // una pila de cadenas
        Pila<String> p3 = new Pila<String>();
        p3.poner("tres");
        p3.poner("dos");
        p3.poner("uno");

        System.out.println( p3.sacar() );
        System.out.println( p3.sacar() );
        System.out.println( p3.sacar() );
    }
}
```

Algunas diferencias más: cuando queremos especializar una clase en algún tipo de datos primitivo como `int` o `double` tenemos que hacerlo pasando sus correspondientes *wrappers*.

12.4.3 Los wrappers (envoltorios) de los tipos de datos primitivos

En Java existen clases "asociadas" a cada uno de los tipos de datos primitivos. A estas clases las llamamos *wrappers*. Veamos la siguiente tabla.

Tipo Primitivo	Wrapper
int	Integer
long	Long
float	Float
double	Double
char	Character
byte	Byte
boolean	Boolean

Fig. 12.3 Tipos de datos y `wrappers` en Java.

Las clases *wrappers* tienen un uso muy particular que iremos viendo en la medida que sea, estrictamente, necesario. Pero como vimos en el ejemplo, el primer uso que le daremos será el de indicar el tipo de datos con el que queremos especializar una clase genérica.

12.4.4 Autoboxing

Llamamos *autoboxing* a la propiedad que permite tratar a los tipos primitivos como si fueran objetos de sus clases *wrappers* y viceversa.

Por ejemplo, en el programa anterior definimos una pila para apilar valores enteros e hicimos lo siguiente:

```
Pila<Integer> p1 = new Pila<Integer>();
p1.poner(3);
p1.poner(2);
p1.poner(1);
```

Si bien la pila está especializada en `Integer`, al momento de utilizarla apilamos los valores literales 3, 2 y 1, todos de tipo `int`.

También podríamos haberlo hecho de la siguiente manera, aunque no tendría ningún sentido:

```
Pila<Integer> p1 = new Pila<Integer>();
p1.poner(new Integer(3));
p1.poner(new Integer(2));
p1.poner(new Integer(1));
```

12.5 Resumen

En este capítulo hicimos la migración que nos permitió pasar de programar en C a Java haciendo una breve escala en C++.

Estudiamos conceptos básicos de la teoría de objetos como encapsulamiento, clases y objetos, métodos, variables de instancia, etcétera.

También analizamos la manera de hacer que una clase sea genérica de forma tal que los tipos de datos de sus miembros sean parametrizables. En C++ esto se hace con *templates*. En Java simplemente lo llamamos *generics*.

En el próximo capítulo estudiaremos el lenguaje Java y analizaremos ejemplos simples para comparar Java con C y C++. El lector se sorprenderá al ver la gran cantidad de coincidencias que existen entre estos lenguajes.

12.6 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

12.6.1 Mapa conceptual

12.6.2 Autoevaluaciones

12.6.3 Presentaciones*

13

Introducción al lenguaje de programación Java

Contenido

Objetivos del capítulo

- Aprender el lenguaje de programación Java
- Comparar el lenguaje Java con C y C++; su sintaxis, tipos, operadores, etcétera.
- Desarrollar ejemplos que le permitan al alumnos familiarizarse con Java.

Competencias específicas

- Comprender, describir y modelar los conceptos principales del paradigma de programación orientado a objetos y aplicarlos a situaciones de la vida real.

13.1 Introducción

Java es un lenguaje de programación de propósitos generales. que permite desarrollar cualquier tipo de aplicación, igual que C, Pascal, etcétera.

Habitualmente, tendemos a asociar su nombre al desarrollo de páginas de Internet. Sin embargo, la creencia popular de que Java es un lenguaje para programar páginas Web es totalmente falsa. La confusión surge porque Java permite "incrustar" programas dentro de las páginas Web para que sean ejecutados dentro del navegador del usuario. Estos son los famosos *applets*, que fueron muy promocionados durante los noventa pero que hoy en día son obsoletos y, prácticamente, quedaron en desuso.

Tampoco debemos confundir Java con JavaScript. El primero es el lenguaje de programación que estudiaremos en este capítulo. El segundo es un lenguaje de *scripting* que permite agregar funcionalidad dinámica en las páginas Web. Nuevamente, la similitud de los nombres puede aportar confusión, pero vale la pena aclarar que JavaScript no tiene nada que ver con Java: son dos cosas totalmente diferentes.

El lenguaje de programación Java se caracteriza por dos puntos bien definidos:

- Es totalmente orientado a objetos.
- Su sintaxis es casi idéntica a la de C++.

Más allá de esto, debemos mencionar que incluye una extensa biblioteca (árbol de clases) que provee funcionalidad para casi todo lo que un programador pueda necesitar. Esto abarca desde manejo de cadenas de caracteres (con la clase `String`) hasta *sockets* (redes, comunicaciones), interfaz gráfica, acceso a bases de datos, etc.

13.2 Comencemos a programar

Nuestro primer programa simplemente mostrará la frase "`Hola Mundo !!!`" en la consola.

```
package libro.cap13;

public class HolaMundo
{
   public static void main(String[] args)
   {
      System.out.println("Hola Mundo !!!");
   }
}
```

Antes de pasar a analizar el código del programa, debo solicitarle al lector que sea paciente y me permita pasar por alto la explicación de algunas de las palabras o sentencias que utilizaré en los ejemplos de este capítulo, como es el caso de las palabras `public`, `static`, `class`, `package`, etcétera. Todas estas serán explicadas en detalle durante el capítulo de programación orientada a objetos.

Ahora sí repasemos el código de nuestro programa: un programa Java es una clase (`class`) que contiene el método (o función) `main`. Este método tiene que ser definido con los modificadores `public static void` y debe recibir un `String[]` como parámetro.

Los bloques de código se delimitan con `{ }` (llaves) y las sentencias finalizan con `;` (punto y coma).

Podemos ver también que el programa comienza declarando un `package`. Por último, con `System.out.println` escribimos en la consola el texto que queremos mostrar.

En Java siempre codificamos clases y cada clase debe estar contenida dentro de un archivo de texto con el mismo nombre que la clase y con extensión `.java`. Así, nuestro programa debe estar codificado en un archivo llamado `HolaMundo.java` (respetando mayúsculas y minúsculas).



Cada clase debe estar contenida dentro de un archivo de texto con el mismo nombre que la clase y con extensión .java.

13.2.1 El Entorno Integrado de Desarrollo (IDE)

Si bien podemos escribir nuestro código utilizando cualquier editor de texto y compilarlo en línea de comandos, lo recomendable es utilizar una herramienta que nos ayude durante todo el proceso de programación.

Como ya sabemos, una IDE es una herramienta que permite editar programas, compilarlos, depurarlos, documentarlos, ejecutarlos, etcétera. Las principales IDE para desarrollar programas Java son Eclipse y NetBeans. Aquí utilizaremos Eclipse.

Durante este capítulo analizaremos una serie de programas simples que nos permitirán comprender el lenguaje Java mediante ejemplos y comparaciones con C y C++.



Instalar y utilizar Eclipse para Java

13.2.2 Entrada y salida estándar

En Java, los objetos `System.in` y `System.out` representan la entrada y la salida estándar respectivamente que, por defecto, son el teclado y la pantalla (en modo texto).

Para escribir en la consola, utilizaremos `System.out.println`. Para leer datos que se ingresan por teclado, utilizaremos objetos de la clase `Scanner` construidos a partir de `System.in` (*standard input*).

En el siguiente programa, le pedimos al usuario que ingrese su nombre para, luego, emitir un saludo en la consola.

```
package libro.cap13;

import java.util.Scanner;

public class HolaMundoNombre
{
   public static void main(String[] args)
   {
      // esta clase permite leer datos por teclado
      Scanner scanner = new Scanner(System.in);

      // mensaje para el usuario
      System.out.print("Ingrese su nombre: ");

      // leemos un valor entero por teclado
      String nom = scanner.nextLine();

      // mostramos un mensaje y luego el valor leido
      System.out.println("Hola Mundo: " + nom);
   }
}
```

La clase `Scanner` permite leer datos a través del teclado. Luego mostramos un mensaje indicando al usuario que debe ingresar su nombre. A continuación, leemos el nombre que



La diferencia entre `System.out.print` y `System.out.println` radica en que el primero imprime en la consola el valor del argumento que le pasamos, mientras que el segundo, además, agrega un salto de línea al final.

el usuario ingrese y lo asignamos en la variable `nom`. Por último, mostramos un mensaje compuesto por un texto literal, `"Hola Mundo: "`, seguido del valor de la variable `nom`.

Notemos la diferencia entre `System.out.print` y `System.out.println`. El primero imprime en la consola el valor del argumento que le pasamos. El segundo, además, agrega un salto de línea al final.

Ahora, veamos un ejemplo más completo donde leemos por teclado datos de diferentes tipos, los concatenamos y mostramos la salida en una única línea.

```
package demo;

import java.util.Scanner;

public class IngresaDatos
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      // ingresamos nombre (tipo String)
      System.out.print("Ingrese su nombre: ");
      String nom = scanner.nextLine();

      // ingresamos edad (tipo int)
      System.out.print("Ingrese su edad: ");
      int edad = scanner.nextInt();

      // ingresamos altura(tipo double)
      System.out.print("Ingrese su altura: ");
      double altura = scanner.nextDouble();

      // concatenamos los datos en una cadena
      String x="";
      x += "Yo soy "+nom+" tengo "+edad+" anios ";
      x += "y mido "+altura+" metros";

      // mostramos por consola
      System.out.println(x);
   }
}
```

13.2.3 Comentarios en el código fuente

Java permite los mismos tipos de comentarios que C y C++.

Comentarios de una sola línea:

```
// esto es una linea de codigo comentada
```

Comentarios de más de una línea:

```
/*
Estas son varias
lineas de codigo
comentadas
*/
```

13.3 Tipos de datos, operadores y estructuras de control

Las estructuras de control, los operadores aritméticos, relacionales y lógicos, unarios y binarios, y los tipos de datos que provee Java, son idénticos a los de C y C++ que estudiamos en los primeros capítulos de este libro.

Es decir, el `while`, `do-while`, `for`, `if`, `switch` e *if-inline*, los tipos `short`, `int`, `long`, `float`, `double` y `char` y los operadores `+`, `-`, `*`, `/`, `%`, `++`, `--`, `+=`, `-=`, `*=`, `/=`, `>`, `<`, `!=`, `<=`, `>=`, etcétera, coinciden totalmente con los que usamos en C.

Java, además, agrega los siguientes tipos de datos: `boolean` y `byte`. El primero admite los valores lógicos `true` y `false`. El segundo admite valores enteros signados de hasta 1 *byte* de precisión.

13.3.1 El bit de signo para los tipos de datos enteros

En Java no existe el modificador *unsigned* de C. Todos los tipos enteros son signados, salvo el `char` que, además, se representa en 2 *bytes*.

13.3.2 El compilador y la máquina virtual (JVM o JRE)

Las aplicaciones Java se ejecutan dentro de una máquina virtual o *runtime enviroment*, conocidos como JVM o JRE. Cuando compilamos un programa el compilador genera un archivo con extensión `.class` que contiene código de máquina virtual, también llamado *bytecode*.

La máquina virtual se instala sobre el sistema operativo y define una plataforma homogénea sobre la que podemos ejecutar nuestras aplicaciones.

Por lo anterior, todos los programas Java corren sobre la misma plataforma (virtual), en la cual las longitudes de los tipos de datos son las siguientes:

Tipo	Descripción	Longitud
`byte`	entero con signo	1 *byte*
`char`	entero sin signo	2 *bytes*
`short`	entero con signo	2 *bytes*
`int`	entero con signo	4 *bytes*
`long`	entero con signo	8 *bytes*
`float`	punto flotante	4 *bytes*
`double`	punto flotante	8 *bytes*
`boolean`	lógico (admite `true` o `false`)	1 *byte*
`String`	objeto, representa una cadena de caracteres	

Fig. 13.1 Longitudes de los tipos de dato.

Analizaremos algunos ejemplos que nos permitan ilustrar el uso de los operadores y las estructuras de control.

13.3.3 Estructuras de decisión

Como ya sabemos, Java tiene las mismas estructuras de decisión que C. Veamos entonces como usar el `if`, `switch` e *if-inline*.

Ejemplo: ¿Es mayor de 21 años?

En el siguiente ejemplo, utilizamos un `if` para determinar si el valor (`edad`) ingresado por el usuario es mayor o igual que 21.

```
package libro.cap13;

import java.util.Scanner;

public class EsMayorQue21
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Ingrese su edad: ");
      int edad = scanner.nextInt();

      if( edad >= 21 )
      {
         System.out.println("Ud. es mayor de edad !");
      }
      else
      {
         System.out.println("Ud. es es menor de edad");
      }
   }
}
```

Ejemplo: ¿Es par o impar?

El siguiente programa pide al usuario que ingrese un valor entero e indica si el valor ingresado es par o impar.

Recordemos que un número es par si es divisible por 2; es decir que el valor residual en dicha división debe ser cero. Para esto, utilizaremos el operador `%` (operador módulo, retorna el residuo de la división).

```
package libro.cap13;

import java.util.Scanner;

public class ParOImpar
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Ingrese un valor: ");
      int v = scanner.nextInt();

      // obtenemos el residuo de dividir v por 2
      int resto = v%2;

      // para preguntar por igual utilizamos ==
      if( resto==0 )
      {
         System.out.println(v+" es par");
```

```
      }
      else
      {
         System.out.println(v+" es impar");
      }
   }
}
```

Ejemplo: ¿Es par o impar? (utilizando un *if-inline*)

```
package libro.cap13;

import java.util.Scanner;

public class ParOImpar2
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Ingrese un valor: ");
      int v = scanner.nextInt();

      // obtenemos el residuo de dividir v por 2
      int resto = v%2;

      // utilizando un if in-line
      String mssg = (resto == 0) ? "es Par": "es Impar";

      // mostramos resultado
      System.out.println(v+" "+mssg);
   }
}
```

Ejemplo: Muestra el día de la semana.

En el siguiente programa, le pedimos al usuario que ingrese un día de la semana (entre 1 y 7) y le mostramos el nombre del día ingresado. Si introduce cualquier otro valor emitimos un mensaje informando que el dato ingresado es incorrecto.

```
package libro.cap13;

import java.util.Scanner;

public class DemoSwitch
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Ingrese un dia de la semana (1 a 7): ");
      int v = scanner.nextInt();

      String dia;
```

```
        switch( v )
        {
           case 1:
              dia = "Lunes";
              break;
           case 2:
              dia = "Martes";
              break;
           case 3:
              dia = "Miercoles";
              break;
           case 4:
              dia = "Jueves";
              break;
           case 5:
              dia = "Viernes";
              break;
           case 6:
              dia = "Sabado";
              break;
           case 7:
              dia = "Domingo";
              break;
           default:
              dia = "Dia incorrecto... Ingrese un valor entre 1 y 7.";
        }

        System.out.println(dia);
     }
  }
```

13.3.4 Estructuras iterativas

Veamos ahora ejemplos que ilustren el uso del `while`, `do-while` y `for`.

Ejemplo: Muestra números naturales.

El siguiente programa utiliza un `while` para mostrar los primeros *n* números naturales, siendo *n* un valor que ingresará el usuario por teclado.

```
package libro.cap13;

import java.util.Scanner;

public class PrimerosNumeros1
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      // leemos el valor de n
      int n = scanner.nextInt();

      int i = 1;
```

```
      while( i<=n )
      {
         // mostramos el valor de i
         System.out.println(i);

         // incrementamos el valor de i
         i++;
      }
   }
}
```

Ejemplo: Muestra números naturales (utilizando `do-while`).

```
package libro.cap13;

import java.util.Scanner;

public class PrimerosNumeros2
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);
      int n = scanner.nextInt();

      int i = 1;

      do
      {
         System.out.println(i);
         i++;
      }
      while( i<=n );
   }
}
```

Ejemplo: Muestra números naturales (utilizando `for`).

```
package libro.cap13;

import java.util.Scanner;

public class PrimerosNumeros3
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);
      int n = scanner.nextInt();

      for( int i=1; i<=n; i++ )
      {
         System.out.println(i);
      }
   }
}
```



En Java existe el tipo `boolean` que admite los valores lógicos `true` y `false`. Por otro lado, y a diferencia de C, el tipo `int` no tiene valor de verdad.

13.3.5 El tipo de datos `boolean` y las expresiones lógicas

En Java existe el tipo `boolean` que admite los valores lógicos `true` y `false`. Por otro lado, y a diferencia de C, el tipo `int` no tiene valor de verdad.

Código C	Código Java

`//:`
`int fin=0;`
 	
`while( !fin )`
`{`
`   //:`
`}`
`//:`
	`//:`
	`boolean fin=false;`
	
	`while( !fin )`
	`{`
	`   //:`
	`}`
	`//:`

Fig. 13.2 Uso de variables booleanas, comparación entre C y Java.

En la tabla anterior, mostramos y comparamos la forma en la que se utilizan las variables booleanas en los lenguajes C y Java. En C los enteros tienen valor de verdad; en cambio, en Java solo tienen valor de verdad las variables de tipo `boolean`.

Lo anterior, más allá de las cuestiones semánticas, hace más robusto al lenguaje de programación ya que algunos de los errores lógicos típicos que cometemos cuando programamos en C, en Java se detectan como errores de compilación.

Para ilustrar esto, veamos el siguiente ejemplo:

Código C	Código Java

`//:`
`int a=3;`
 	
`if( a=10 )`
`{`
`   printf("a vale 10\n");`
`}`
`else`
`{`
`   printf("a no es 10\n");`
`}`
`//:`
	`//:`
	`int a=3;`
	
	`if( a=10 )`
	`{`
	`   System.out.println("a vale 10");`
	`}`
	`else`
	`{`
	`   System.out.println("a no es 10");`
	`}`
	`//:`

Como se puede apreciar a simple vista, la codificación en ambos lenguajes es idéntica. Tanto en Java como en C el operador `==` (igual igual) es el operador de comparación mientras que el operador `=` (igual) es el operador de asignación.

Los dos lenguajes permiten escribir código compacto. Entonces, resulta que la sentencia:

```
if( a=10 )
{
   // :
}
else
{
   // :
}
```

primero asigna 10 a la variable `a` y luego pregunta por el valor de verdad de la expresión encerrada dentro de los paréntesis del `if`.

En C, este programa siempre entra por el `if`, nunca por el `else`, porque como `a` vale 10 (distinto de cero) su valor de verdad es verdadero. Esto es un error de lógica que se manifestará en tiempo de ejecución ya que lo que realmente queríamos hacer era mostrar un mensaje en función del valor numérico de la variable `a`.

En Java el mismo código dará un error en tiempo de compilación porque luego de asignarle 10 a la variable `a`, se evaluará el valor de verdad de la expresión encerrada dentro de los paréntesis y, como dijimos más arriba, los enteros no tienen valor de verdad.

13.3.6 Las constantes

Las constantes se definen fuera de los métodos utilizando el modificador `final`. Habitualmente, se las define como públicas y estáticas (`public static`).

Ejemplo: Muestra día de la semana (utilizando constantes).

```
package libro.cap13;

import java.util.Scanner;

public class DemoConstantes
{
   // definimos las constantes
   public static final int LUNES = 1;
   public static final int MARTES = 2;
   public static final int MIERCOLES = 3;
   public static final int JUEVES = 4;
   public static final int VIERNES = 5;
   public static final int SABADO = 6;
   public static final int DOMINGO = 7;

   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Ingrese un dia de la semana (numero): ");
      int v = scanner.nextInt();

      String dia;

      switch( v )
      {
         case LUNES:
            dia = "Lunes";
            break;
         case MARTES:
            dia = "Martes";
            break;
         case MIERCOLES:
            dia = "Miercoles";
            break;
         case JUEVES:
            dia = "Jueves";
            break;
         case VIERNES:
            dia = "Viernes";
            break;
```

```
         case SABADO:
            dia = "Sabado";
            break;
         case DOMINGO:
            dia = "Domingo";
            break;
         default:
            dia = "Dia incorrecto... Ingrese un valor entre 1 y 7.";
      }

      System.out.println(dia);
   }
}
```

13.3.7 Arrays

En Java, al igual que en C, los *arrays* comienzan siempre desde cero.

```
// definimos un array de 10 elementos enteros numerados de 0 a 9
int arr[] = new int[10];
```

También podemos construir un *array* de *n* elementos, siendo *n* una variable.

```
int n = 10;
int arr[] = new int[n];
```

Ejemplo: Almacena valores en un *array*.

En el siguiente ejemplo, definimos un *array* de 10 enteros. Luego pedimos al usuario que ingrese valores numéricos (no más de diez) y los guardamos en el *array*. Por último, recorremos el *array* para mostrar su contenido.

```
package libro.cap13;

import java.util.Scanner;

public class DemoArray
{
   public static void main(String[] args)
   {
      // definimos un array de 10 enteros
      int arr[] = new int[10];

      // el scanner para leer por teclado...
      Scanner scanner = new Scanner(System.in);

      // leemos el primer valor
      System.out.print("Ingrese un valor (0=>fin): ");
      int v = scanner.nextInt();

      int i=0;
```

```
      // mientras v sea distinto de cero AND i sea menor que 10
      while( v!=0 && i<10 )
      {
         // asignamos v en arr[i] y luego incrementamos el valor de i
         arr[i++] = v;

         // leemos el siguiente valor
         System.out.print("Ingrese el siguiente valor (0=>fin): ");
         v = scanner.nextInt();
      }

      // recorremos el array mostrando su contenido
      for( int j=0; j<i; j++ )
      {
         System.out.println(arr[j]);
      }
   }
}
```

Si conocemos de antemano los valores que vamos a almacenar en el *array* entonces podemos definirlo “por extensión”. Esto crea el *array* con la dimensión necesaria para contener el conjunto de valores y asigna los elementos del conjunto en posiciones consecutivas del *array*.

```
// creamos un array de strings con los nombres de Los Beatles
String beatles[] = {"John", "Paul", "George", "Ringo"};
```

En Java los *arrays* son objetos. Tienen el atributo `length` que indica su capacidad.

```
// imprimimos en consola cuantos son Los Beatles
System.out.println("Los Beatles son: "+ beatles.length);
```



En Java los *arrays* son objetos. Tienen el atributo `length` que indica su capacidad.

Las siguientes son formas correctas y equivalentes para declarar *arrays*:

```
String arr[];
String []arr;
String[] arr;
```

Notemos también que no es lo mismo “declarar” un *array* que “crearlo o instanciarlo”.

```
// definimos un array de strings (aun sin dimensionar)
String arr[];

// creamos (instanciamos) el array
arr = new String[10];
```

o bien

```
// definimos e instanciamos el array de 10 strings
String arr[] = new String[10]
```

Ejemplo: Muestra día de la semana (utilizando un *array*).

```
package libro.cap13;

import java.util.Scanner;

public class DemoArray2
{
   public static void main(String[] args)
   {
      String dias[] = {"Lunes"  , "Martes", "Miercoles", "Jueves"
                      ,"Viernes", "Sabado", "Domingo"};

      Scanner scanner = new Scanner(System.in);

      System.out.print("Ingrese un dia de la semana (numero): ");
      int v = scanner.nextInt();

      if( v<=dias.length )
      {
         // recordemos que los arrays se numeran desde cero
         System.out.println( dias[v-1] );
      }
      else
      {
         System.out.println("Dia incorrecto...");
      }
   }
}
```

■

13.3.8 Matrices

Una matriz es un *array* de dos dimensiones. Se define de la siguiente manera:

```
// definimos una matriz de enteros de 3 filas por 4 columnas
int mat[][] = new int[3][4];
```

Ejemplo: Llena una matriz con números aleatorios.

En el siguiente programa, pedimos al usuario que ingrese las dimensiones de una matriz (filas y columnas). Luego creamos una matriz con esas dimensiones y la llenamos con números generados aleatoriamente.

```
package libro.cap13;

import java.util.Scanner;

public class DemoMatriz
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Ingrese cantidad de filas: ");
      int n = scanner.nextInt();
```

```
      System.out.print("Ingrese cantidad de columnas: ");
      int m = scanner.nextInt();

      // creamos una matriz de n filas x m columnas
      int mat[][] = new int[n][m];

      for(int i=0; i<n; i++ )
      {
         for(int j=0; j<m; j++)
         {
            // generamos un numero aleatorio entre 0 y 1000
            int nro = (int)(Math.random()*1000);

            // asignamos el numero en la matriz
            mat[i][j] = nro;
         }
      }

      for(int i=0; i<n; i++ )
      {
         for(int j=0; j<m; j++)
         {
            // imprimimos la celda de la matriz
            System.out.print(mat[i][j]+"\t");
         }
         System.out.println();
      }
   }
}
```
■

En este ejemplo utilizamos `Math.random` para generar un número aleatorio. El método `random` de la clase `Math` genera un número mayor que cero y menor que 1. Lo multiplicamos por 1000 y luego lo convertimos a `int` para obtener la parte entera.

Notemos también que, casi al final, donde imprimimos cada celda de la matriz con sentencia:

```
System.out.print(mat[i][j]+"\t");
```

concatenamos un carácter especial: `\t` (léase “barra te”). Este carácter representa al tabulador. Es decir que luego de mostrar el contenido de cada celda imprimimos un tabulador para que los números de la matriz se vean alineados.



Utilizamos `Math.random` para generar un número aleatorio. El método `random` de la clase `Math` genera un número mayor que cero y menor que 1.

La salida de este programa será la siguiente:

```
Ingrese cantidad de filas: 3
Ingrese cantidad de columnas: 4
714 529 331 200
580 25  374 12
554 345 979 620
```

Dijimos que una matriz es un *array* de dos dimensiones, pero también podríamos decir que una matriz es un *array* de *arrays*. Viéndolo de esta manera entonces podemos conocer la cantidad de filas y columnas de una matriz a través del atributo `length`.

```
int [][]mat = new int[5][3];
int filas  = mat.length;     // cantidad de filas
int colums = mat[0].length; // cantidad de columnas
```

Se puede inicializar una matriz definiendo sus valores por extensión como veremos en el siguiente ejemplo.

```
int mat[][] = { {3, 2, 1}
               ,{5, 3, 7}
               ,{1, 9, 2}
               ,{4, 6, 5} };
```

Esto dimensiona la matriz `mat` con 4 filas y 3 columnas y además asigna los valores en las celdas correspondientes.

13.3.9 Literales de cadenas de caracteres

Una cadena de caracteres literal se representa encerrada entre comillas dobles, por ejemplo: `"Esto es una cadena"`. En cambio, un carácter literal se representa encerrado entre comillas simples, por ejemplo: `'A'`.

Las cadenas son tratadas como objetos, los caracteres (al igual que en C) son valores numéricos enteros.

En Java las cadenas son tratadas como objetos; por lo tanto, `"Esto es una cadena"` es un objeto sobre el cual podemos invocar métodos, como veremos a continuación:

```
// imprimimos ESTO ES UNA CADENA (en mayusculas)
System.out.println("Esto es una cadena".toUpperCase());
```

En cambio, los caracteres (al igual que en C) son valores numéricos enteros. Por ejemplo, `'A'` es en realidad el valor 65 (el código ASCII del carácter).

Notemos además que no es lo mismo `"A"` que `'A'`. El primero es una cadena de caracteres que contiene un único carácter; es un objeto. El segundo es un valor literal de tipo `char`, un valor numérico.

Veamos el siguiente ejemplo:

```
package libro.cap13;

public class DemoCaracteres
{
   public static void main(String[] args)
   {
      for( int i=0; i<5; i++ )
      {
         System.out.println(i+"A");
      }
   }
}
```

En este ejemplo concatenamos al valor numérico de `i` la cadena `"A"`; entonces la salida del programa será la siguiente:

```
0A
1A
2A
3A
4A
```

Si en lugar de concatenar `"A"` hubiéramos concatenado `'A'`, así:

```
System.out.println(i+'A');
```

la salida hubiera sido la siguiente:

```
65
66
67
68
69
```

Esto se debe a que `'A'` es, en realidad, el valor numérico 65; entonces en cada iteración del `for` imprimimos `i+65`, comenzando con `i` igual a 0.

Le recomiendo al lector pensar cuál será la salida del siguiente programa:

```
package libro.cap13;

public class DemoCaracteres2
{
   public static void main(String[] args)
   {
      for( int i='A'; i<='Z'; i++ )
      {
         System.out.println(i);
      }
   }
}
```

Probablemente, el lector espere una salida como esta:

```
A
B
C
:
Z
```

Sin embargo, la salida sera:

```
65
66
67
:
90
```

¿Por qué? Porque lo que estamos imprimiendo es el valor de `i`, que es una variable `int` cuyo valor inicial será 65 y se incrementará hasta llegar a 90 (código ASCII del carácter 'Z'). Por lo tanto, `System.out.println` imprime su valor numérico. Claro que esto se puede arreglar convirtiendo el valor de `i` a `char` como vemos a continuación.

```
package libro.cap13;

public class DemoCaracteres3
{
   public static void main(String[] args)
   {
      char c;
      for( int i='A'; i<='Z'; i++ )
      {
         // para asignar un int en un char debemos "castear"
         c = (char) i;
         System.out.println(c);
      }
   }
}
```



En Java existen los mismos caracteres especiales que en C. Estos caracteres se pueden utilizar anteponiendo la barra \ (léase “barra” o carácter de “escape”).

13.3.10 Caracteres especiales

En Java existen los mismos caracteres especiales que en C. Estos caracteres se pueden utilizar anteponiendo la barra \ (léase “barra” o carácter de “escape”). Algunos son:

`\t` – tabulador
`\n` – salto de línea
`\"` – comillas dobles
`\'` – comillas simples
`\\` – barra

Veamos un ejemplo:

```
package libro.cap13;

public class DemoCaracteresEspeciales
{
   public static void main(String[] args)
   {
      System.out.println("Esto\t es un \"TABULADOR\"");
      System.out.println("Esto es un\nBARRA\nENE\n\n :O)");
      System.out.println("La barra es asi: \\");
   }
}
```

La salida de este programa será la siguiente:

```
Esto     es un "TABULADOR"
Esto es un
BARRA
ENE

 :O)
La barra es asi: \
```

13.3.11 Argumentos en línea de comandos

En Java accedemos a los argumentos pasados en la línea de comandos a través del `String[]` que recibe el método `main`.

Ejemplo: Muestra los argumentos pasados a través de la línea de comandos.

```
package libro.cap13;

public class EchoJava
{
   public static void main(String[] args)
   {
      for(int i=0; i<args.length; i++)
      {
         System.out.println(args[i]);
      }

      System.out.println("Total: "+args.length+" argumentos.");
   }
}
```

Si ejecutamos este programa desde la línea de comandos así:

```
c:\>java EchoJava Hola que tal?
```

la salida será:

```
Hola
que
tal?
Total: 3 argumentos.
```

El lector podrá acceder al videotutorial donde se muestra cómo pasar argumentos en línea de comandos a un programa que se ejecuta desde Eclipse.

13.4 Tratamiento de cadenas de caracteres

Como vimos anteriormente, las cadenas de caracteres son tratadas como objetos porque, en Java, `String` es una clase.

La clase `String` define métodos que permiten manipular los caracteres de las cadenas. Algunos de estos métodos los explicaremos a continuación.



Las cadenas de caracteres, variables o literales son tratadas como objetos; esto se debe a que, en Java, `String` es una clase.

13.4.1 Acceso a los caracteres de un string

Una cadena representa una secuencia finita de cero o más caracteres numerados a partir de cero. Es decir que la cadena `"Hola"` tiene 4 caracteres numerados entre 0 y 3.

Ejemplo: Acceso directo a los caracteres de una cadena.

```
package libro.cap13.cadenas;

public class Cadenas
{
   public static void main(String[] args)
   {
      String s = "Esta es mi cadena";

      System.out.println( s.charAt(0) );
      System.out.println( s.charAt(5) );
      System.out.println( s.charAt(s.length()-1) );

      for(int i=0; i<s.length(); i++)
      {
         char c = s.charAt(i);
         System.out.println(i+" -> "+c);
      }
   }
}
```



No debemos confundir el atributo `length` de los *array* con el método `length` de los *string*. En el caso de los *array*, por tratarse de un atributo no lleva paréntesis. En cambio, en la clase `String`, `length` es un método; por lo tanto, siempre debe invocarse con paréntesis.

El método `charAt` retorna el carácter (tipo `char`) ubicado en una posición determinada. El método `length` retorna la cantidad de caracteres que tiene la cadena.

No debemos confundir el atributo `length` de los *array* con el método `length` de los *string*. En el caso de los *array*, por tratarse de un atributo no lleva paréntesis. En cambio, en la clase `String`, `length` es un método; por lo tanto, siempre debe invocarse con paréntesis. Veamos:

```
char c[] = { 'H', 'o', 'l', 'a' };
System.out.println( c.length );

String s = "Hola";
System.out.println( s.length() );
```

13.4.2 Mayúsculas y minúsculas

Ejemplo: Pasar una cadena a mayúsculas y a minúsculas.

```
package libro.cap13.cadenas;

public class Cadenas1
{
   public static void main(String[] args)
   {
      String s = "Esto Es Una Cadena de caRACtERes";
      String sMayus = s.toUpperCase();
      String sMinus = s.toLowerCase();

      System.out.println("Original: "+s);
      System.out.println("Mayusculas: "+sMayus);
      System.out.println("Minusculas: "+sMinus);
   }
}
```

Recordemos que `s` es un objeto que contiene información (la cadena en sí misma) y los métodos necesarios para manipularla: entre otros, los métodos `toUpperCase` y

`toLowerCase` que utilizamos en este ejemplo para retornar dos nuevas cadenas con los mismos caracteres que la cadena original pero en mayúsculas o en minúsculas según corresponda.

13.4.3 Ocurrencias de caracteres

Ejemplo: Ubicar la posición de un carácter dentro de una cadena.

```
package libro.cap13.cadenas;

public class Cadenas2
{
   public static void main(String[] args)
   {
      String s = "Esto Es Una Cadena de caRACtERes";

      int pos1 = s.indexOf('C');
      int pos2 = s.lastIndexOf('C');
      int pos3 = s.indexOf('x');

      System.out.println(pos1);
      System.out.println(pos2);
      System.out.println(pos3);
   }
}
```

El método `indexOf` retorna la posición de la primera ocurrencia de un carácter dentro del *string*. Si la cadena no contiene ese carácter entonces retorna un valor negativo. Análogamente, el método `lastIndexOf` retorna la posición de la última ocurrencia del carácter dentro del *string* o un valor negativo en caso de que el carácter no esté contenido dentro de la cadena.

13.4.4 Subcadenas

Ejemplo: Uso del método `substring` para obtener porciones de la cadena original.

```
package libro.cap13.cadenas;

public class Cadenas3
{
   public static void main(String[] args)
   {
      String s = "Esto Es Una Cadena de caRACtERes";

      String s1 = s.substring(0,7);
      String s2 = s.substring(8,11);

      // tomamos desde el caracter 8 hasta el final
      String s3 = s.substring(8);

      System.out.println(s1);
      System.out.println(s2);
      System.out.println(s3);
   }
}
```

La salida de este programa es la siguiente:

```
Esto Es
Una
Una Cadena de caRACtERes
```

El método `substring` puede invocarse pasándole dos argumentos o uno solo. Si lo invocamos con dos argumentos estaremos indicando las posiciones *desde* (inclusive) y *hasta* (no inclusive) que delimitarán la subcadena que queremos extraer. En cambio, si lo invocamos con un solo argumento estaremos indicando que la subcadena por extraer comienza en la posición especificada (inclusive) y se extenderá hasta el final del *string*.

13.4.5 Prefijos y sufijos

Con los métodos `startWith` y `endsWith`, podemos verificar fácilmente si una cadena comienza con un determinado prefijo o termina con algún sufijo.

```
package libro.cap13.cadenas;

public class Cadenas4
{
   public static void main(String[] args)
   {
      String s = "Un buen libro de Algoritmos";

      boolean b1 = s.startsWith("Un buen");  // true
      boolean b2 = s.startsWith("A");        // false
      boolean b3 = s.endsWith("Algoritmos"); // true
      boolean b4 = s.endsWith("Chau");       // false

      System.out.println(b1);
      System.out.println(b2);
      System.out.println(b3);
      System.out.println(b4);
   }
}
```

13.4.6 Posición de un substring dentro de la cadena

Los métodos `indexOf` y `lastIndexOf` están sobrecargados de forma tal que permiten detectar la primera y la última ocurrencia (respectivamente) de un *substring* dentro de la cadena.

```
package libro.cap13.cadenas;

public class Cadenas5
{
   public static void main(String[] args)
   {
      String s = "Un buen libro de Algoritmos, un buen material";

      int pos1 = s.indexOf("buen");     // retorna 3
      int pos2 = s.lastIndexOf("buen"); // retorna 32
      System.out.println(pos1);
      System.out.println(pos2);
   }
}
```

13.4.7 Conversión entre números y cadenas

A continuación, veremos como realizar conversiones entre valores numéricos y cadenas y viceversa.

Algunos ejemplos son:

```
// --- operaciones con el tipo int ---
int i = 43;

// convertimos de int a String
String sInt = Integer.toString(i);

// convertimos de String a int
int i2 = Integer.parseInt(sInt);

// --- operaciones con el tipo double ---
double d = 24.2;

// convertimos de double a String
String sDouble = Double.toString(d);

// convertimos de String a double
double d2 = Double.parseDouble(sDouble);
```

13.4.8 Representación en diferentes bases numéricas

Java, igual que C, permite expresar valores enteros en base 8 y en base 16. Para representar un entero en base 16 debemos anteponerle el prefijo `0x` (léase "cero equis").

```
int i = 0x24ACF;       // en decimal es 150223
System.out.println(i); // imprime 150223
```

Para expresar enteros en base 8 debemos anteponerles el prefijo `0` (léase "cero").

```
int j = 0537;          // en decimal es 351
System.out.println(j); // imprime 351
```

Utilizando la clase `Integer` podemos obtener la representación binaria, octal, hexadecimal y cualquier otra base numérica de un valor entero especificado. Esto lo veremos en el siguiente programa.



Utilizando la clase `Integer` podemos obtener la representación binaria, octal, hexadecimal y cualquier otra base numérica de un valor entero especificado

Ejemplo: Muestra un valor entero en diferentes bases numéricas.

```
package libro.cap13.cadenas;

import java.util.Scanner;

public class Cadenas6
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Ingrese un valor entero: ");
      int v = scanner.nextInt();

      System.out.println("Valor Ingresado: "+v);
      System.out.println("binario = "+Integer.toBinaryString(v));
      System.out.println("octal = "+Integer.toOctalString(v));
      System.out.println("hexadecimal = "+Integer.toHexString(v));
```

```
      System.out.print("Ingrese una base numerica: ");
      int b = scanner.nextInt();

      String sBaseN = Integer.toString(v,b);
      System.out.println(v + " en base("+b+") = " + sBaseN);
   }
}
```

Aquí el usuario ingresa un valor numérico por teclado y el programa le muestra sus representaciones binarias, octales y hexadecimales. Luego se le pide al usuario que ingrese cualquier otra base numérica para mostrar, a continuación, la representación del número expresado en la base ingresada por el usuario.

Por ejemplo, si el usuario ingresa el valor 632 y luego ingresa la base 12 entonces la salida del programa será la siguiente:

```
Ingrese un valor entero: 632
Valor Ingresado: 632
binario = 1001111000
octal = 1170
hexadecimal = 278
Ingrese una base numerica: 12
632 en base(12) = 448
```

13.4.9 La clase `StringTokenizer`

La funcionalidad de esta clase la explicaremos sobre el siguiente ejemplo. Sea la cadena `s` definida a continuación:

```
String s  = "Juan|Marcos|Carlos|Matias";
```

Si consideramos como separador al carácter `|` (léase “carácter pipe”) entonces llamaremos *token* a las subcadenas encerradas entre las ocurrencias de dicho carácter y a las subcadenas encerradas entre este y el inicio o el fin de la cadena `s`.

Para hacerlo más simple, el conjunto de *tokens* que surgen de la cadena `s`, considerando como separador al carácter `|`, es el siguiente:

```
tokens = {Juan, Marcos, Carlos, Matias}
```

Pero si en lugar de tomar como separador al carácter `|` consideramos como separador al carácter `a` sobre la misma cadena `s`, el conjunto de *tokens* será:

```
tokens = {Ju, n|M, rcos|C, rlos|M, ti, s};
```

Utilizando la clase `StringTokenizer` podemos separar una cadena en *tokens* delimitados por un separador. En el siguiente ejemplo, veremos cómo hacerlo.

```
package libro.cap13.cadenas;

import java.util.StringTokenizer;

public class Cadenas7
{
   public static void main(String[] args)
   {
      String s = "Juan|Marcos|Carlos|Matias";
      StringTokenizer st = new StringTokenizer(s,"|");
```

```
      String tok;
      while( st.hasMoreTokens() )
      {
         tok = st.nextToken();
         System.out.println(tok);
      }
   }
}
```

Primero instanciamos el objeto `st` pasándole como argumentos la cadena `s` y una cadena `"|"` que será considerada como separador. Luego, el objeto `st` (objeto de la clase `StringTokenizer`) provee los métodos `hasMoreTokens` y `nextToken` que permiten (respectivamente) controlar si existen más *tokens* en la cadena y avanzar al siguiente *token*.

Notemos que el recorrido a través de los *tokens* de la cadena es *forward only*. Es decir, solo se pueden recorrer desde el primero hasta el último (de izquierda a derecha). No se puede tener acceso directo a un *token* en particular ni tampoco se puede retroceder para recuperar el *token* anterior.

13.4.10 Comparación de cadenas

En Java no existe un tipo de datos primitivo para representar cadenas de caracteres. Las cadenas se representan como objetos de la clase `String`.

Ahora bien, dado que las cadenas son objetos y los objetos son punteros, resulta que si comparamos dos cadenas utilizando el operador `==`, lo que realmente estaremos comparando serán las direcciones de memoria a las que apuntan los objetos.



Dado que las cadenas son objetos y los objetos son punteros, resulta que si comparamos dos cadenas utilizando el operador `==`, lo que realmente estaremos comparando serán las direcciones de memoria a las que apuntan los objetos.

Esto podemos verificarlo con el siguiente programa, en el que se le pide al usuario que ingrese una cadena y luego otra. El programa compara ambas cadenas con el operador `==` e informa el resultado obtenido. Le recomiendo al lector probarlo ingresando dos veces la misma cadena y observar el resultado.

```
package libro.cap13.cadenas;

import java.util.Scanner;

public class Cadenas8
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Ingrese una cadena: ");
      String s1 = scanner.next();

      System.out.print("Ingrese otra cadena: ");
      String s2 = scanner.next();

      // mostramos las cadenas para verificar lo que contienen
      System.out.println("s1 = [" + s1 + "]");
      System.out.println("s2 = [" + s2 + "]");
```

```
        // esto esta mal !!!
        if( s1==s2 )
        {
           System.out.println("Son iguales");
        }
        else
        {
           System.out.println("Son distintas");
        }
     }
  }
```



Para comparar cadenas debemos usar el método `equals` que compara sus contenidos y retorna `true` o `false` según estos sean iguales o no.

La salida de este programa siempre será: `Son distintas`.

Lo correcto será comparar las cadenas utilizando el método `equals`. Este método compara los contenidos y retorna `true` o `false` según estos sean iguales o no.

El ejemplo anterior debe modificarse de la siguiente manera:

```
        // :
        // Ahora si !!!
        if( s1.equals(s2) )
        {
           System.out.println("Son iguales");
        }
        // :
```

Veamos este otro caso:

```
package libro.cap13.cadenas;

public class Cadenas9
{
   public static void main(String[] args)
   {
      // dos cadenas iguales
      String s1 = "Hola";
      String s2 = "Hola";

      System.out.println("s1 = [" + s1 + "]");
      System.out.println("s2 = [" + s2 + "]");

      if( s1==s2 )
      {
         System.out.println("Son iguales");
      }
      else
      {
         System.out.println("Son distintas");
      }
   }
}
```

En este caso el operador `==` retorna `true`, es decir, compara "bien" y el programa indicará que ambas cadenas son idénticas. ¿Por qué? Porque Java asigna la cadena literal `"Hola"` en un espacio de memoria y, ante la aparición de la misma cadena, no vuelve a asignar memoria para almacenar la misma información; simplemente obtiene una nueva referencia a dicho espacio. En consecuencia, los dos objetos `s1` y `s2` apuntan al mismo espacio de memoria, son punteros idénticos y por este motivo el operador `==` retorna `true`. El método `equals` también hubiera retornado `true`.

13.5 Resumen

En este capítulo tuvimos una aproximación al lenguaje de programación Java. Desarrollamos una serie de ejemplos a través de los cuales conocimos su sintaxis y semántica, operadores lógicos, aritméticos y relacionales, sus estructuras de control, etcétera.

En el próximo capítulo, estudiaremos en detalle el paradigma de la programación orientada a objetos para poder aprovechar al máximo las ventajas del lenguaje Java.

13.6 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

13.6.1 Mapa conceptual

13.6.2 Autoevaluaciones

13.6.3 Videotutorial

13.6.3.1 Instalar y utilizar Eclipse para Java

13.6.4 Presentaciones*

14

Programación orientada a objetos

Contenido

Objetivos del capítulo

- Estudiar y comprender el paradigma de la programación orientada a objetos.
- Comprender la potencia y la capacidad de abstracción que ofrece el polimorfismo.
- Utilizar UML como lenguaje gráfico y de comunicación visual.
- Aprender a manejar colecciones en Java (*Java Collection Framework*)
- Comprender la importancia de las *interfaces*.
- Implementar factorías de objetos que permitan desacoplar las implementaciones.
- Conocer y desarrollar implementaciones de las *interfaces* `Comparable` y `Comparator`.
- Manejar errores a través del tratamiento y la propagación de excepciones.

Competencias específicas

- Comprender, describir y modelar los conceptos principales del paradigma de programación orientado a objetos y aplicarlos a situaciones de la vida real.
- Implementar clases y objetos cumpliendo las reglas de la programación orientada a objetos.
- Implementar constructores y destructores para inicializar atributos y liberar recursos.
- Sobrecargar métodos y operadores para optimizar el código de una clase.
- Implementar la herencia en clases derivadas para reutilizar los miembros de una clase base.
- Implementar interfaces y clases polimórficas.
- Identificar, manejar, gestionar y crear las condiciones de error que interrumpan el flujo de ejecución de un programa.

14.1 Introducción

Java es un lenguaje fuertemente tipado. Esto significa que para todo recurso que vayamos a utilizar, previamente, debemos definirle su tipo de datos.

Definición 1: llamamos “objeto” a toda variable cuyo tipo de datos es una “clase”.

Definición 2: llamamos “clase” a una estructura que agrupa datos más la funcionalidad necesaria para manipular dichos datos.

Las cadenas de caracteres, por ejemplo, son objetos de la clase `String`; por lo tanto, almacenan información (la cadena en sí misma) y la funcionalidad necesaria para manipularla. Veamos las siguientes líneas de código:

```
String s = "Hola Mundo";
int i = s.indexOf("M");
```

El objeto `s` almacena la cadena `"Hola Mundo"` y tiene la capacidad de informar la posición de la primera ocurrencia de un determinado carácter dentro de la cadena.

14.2 Clases y objetos



Las clases definen la estructura de sus objetos. Es decir que todos los objetos de una misma clase podrán almacenar el mismo tipo de información y tendrán la misma capacidad para manipularla.

Las clases definen la estructura de sus objetos. Es decir que todos los objetos de una misma clase podrán almacenar el mismo tipo de información y tendrán la misma capacidad para manipularla.

Por ejemplo, pensemos en algunas fechas: 4 de junio de 2008, 15 de junio de 1973 y 2 de octubre de 1970. Evidentemente, las tres fechas son diferentes, pero tienen la misma estructura: todas tienen un día, un mes y un año.

Justamente, el día, el mes y el año son los datos que hacen que una fecha sea distinta de otra. Diremos que son sus “atributos”. Una fecha es distinta de otra porque tiene diferentes valores en sus atributos. Aun así, todas son fechas.

A continuación, analizaremos un ejemplo basado en el desarrollo de la clase `Fecha`. Este desarrollo se hará de manera progresiva, agregando nuevos conceptos paso a paso. Por este motivo, le recomendamos al lector no detener la lectura hasta tanto el ejemplo no haya concluido.

Definiremos entonces la clase `Fecha` que nos permitirá operar con fechas en nuestros programas. Recordemos que en Java cada clase debe estar contenida en su propio archivo de código fuente, con el mismo nombre que la clase y con extensión `.java`. En este caso, la clase `Fecha` debe estar dentro del archivo `Fecha.java`.

```
package libro.cap14.fechas;

public class Fecha
{
   private int dia;
   private int mes;
   private int anio;
}
```

A simple vista la clase se ve como un `struct` de C o un `record` de Pascal. Sin embargo, tiene varias diferencias. Para comenzar, los datos están definidos como `private`; esto significa que desde afuera de la clase no podrán ser accedidos porque son privados, están encapsulados y su acceso solo se permitirá estando dentro de la clase.

En otras palabras, el siguiente código no compila:

```
package libro.cap14.fechas;

public class TestFecha
{
   public static void main(String[] args)
   {
      Fecha f = new Fecha();
      f.dia = 2;     // la variable dia es privada, no tenemos acceso
      f.mes = 10;    // idem...
      f.anio = 1970; // olvidalo...
   }
}
```

Al intentar compilar esto, obtendremos un error de compilación que dice: "`The field Fecha.dia is not visible`" (el campo `dia` de la clase `Fecha` no es visible).

Como las variables `dia`, `mes` y `anio` están declaradas con `private`, quedan encapsuladas dentro de la clase `Fecha` y cualquier intento de accederlas por fuera de la clase generará un error de compilación.

La única forma de asignar valores a estas variables será a través de los métodos que la clase `Fecha` provea para hacerlo.



Los métodos de una clase se escriben como funciones. Dentro de los métodos, podemos acceder a los atributos como si fueran variables globales.

14.2.1 Los métodos

Los métodos de una clase se escriben como funciones. Dentro de los métodos, podemos acceder a los atributos como si fueran variables globales.

A continuación, agregaremos a la clase `Fecha` métodos para asignar (*set*) y para obtener (*get*) el valor de sus atributos. A estos métodos se les suele denominar *setters* y *getters* respectivamente.

```
package libro.cap14.fechas;

public class Fecha
{
   private int dia;
   private int mes;
   private int anio;

   public int getDia()
   {
      // retorna el valor de la variable dia
      return dia;
   }

   public void setDia(int dia)
   {
      // asigna el valor del parametro a la variable dia
      this.dia = dia;
   }

   public int getMes()
   {
      return mes;
   }
```

```
    public void setMes(int mes)
    {
       this.mes = mes;
    }

    public int getAnio()
    {
       return anio;
    }

    public void setAnio(int anio)
    {
       this.anio = anio;
    }
}
```

Ahora la clase provee métodos a través de los cuales podemos acceder a los atributos de sus objetos para asignar y/o consultar sus valores, como vemos en el siguiente código:

```
package libro.cap14.fechas;

public class TestFecha
{
    public static void main(String[] args)
    {
       Fecha f = new Fecha();
       f.setDia(2);
       f.setMes(10);
       f.setAnio(1970);

       // imprimimos el dia
       System.out.println("Dia="+f.getDia());

       // imprimimos el mes
       System.out.println("Mes="+f.getMes());

       // imprimimos el anio
       System.out.println("Anio="+f.getAnio());

       // imprimimos la fecha
       System.out.println(f);
    }
}
```

Métodos de acceso
Por convención, deben llamarse *setXxxx* y *getXxxx*, donde *Xxxx* es el atributo al que el método permitirá acceder. Nos referiremos a los métodos de acceso como los "*setters* y *getters*" de los atributos.

En este ejemplo utilizamos los *setters* y *getters*, primero para asignar valores a los atributos de la fecha y luego, para mostrarlos.

A estos métodos se les llama métodos de acceso o *accessor methods* ya que permiten acceder a los atributos de los objetos para asignarles valor (*set*) o para obtener su valor (*get*). Por convención, deben llamarse *setXxxx* y *getXxxx*, donde *Xxxx* es el atributo al que el método permitirá acceder. Genéricamente, nos referiremos a los métodos de acceso como los "*setters* y *getters*" de los atributos.

En el programa imprimimos por separado los valores de los atributos del objeto `f`, pero al final imprimimos el "objeto completo".

Contrariamente a lo que el lector podría esperar, la salida de:

```
System.out.println(f);
```

no será una fecha con el formato que, habitualmente, usamos para representarlas. La salida de esta línea de código será algo así:

```
libro.cap14.fechas.Fecha@360be0
```

¿Por qué? Porque `System.out.println` no puede saber de antemano cómo queremos que imprima los objetos de las clases que nosotros programamos. Para solucionarlo debemos sobrescribir el método `toString` que heredamos de la clase `Object`.



La "herencia" permite definir clases en función de otras clases ya existentes.
En Java, directa o indirectamente, todas las clases heredan de una clase base llamada `Object`.
Por esto, los métodos definidos en `Object` son comunes a todas las otras clases.

14.2.2 Herencia y sobrescritura de métodos

Una de las principales características de la programación orientada a objetos es la "herencia" que permite definir clases en función de otras clases ya existentes. Es decir, una clase define atributos y métodos y, además, hereda los atributos y métodos que define su "padre" o "clase base".

Si bien este tema lo estudiaremos en detalle más adelante, llegamos a un punto en el que debemos saber que en Java, directa o indirectamente, todas las clases heredan de una clase base llamada `Object`. No hay que especificar nada para que esto ocurra. Siempre es así.

La herencia es transitiva. Sean las clases `A`, `B` y `C`, si `A` hereda de `B` y `B` hereda de `C`, entonces `A` hereda de `C`.

Pensemos en las clases `Empleado` y `Persona`. Evidentemente, un empleado primero es una persona ya que este tendrá todos los atributos de `Persona` (que podrían ser `nombre`, `fechaNacimiento`, `DNI`, etc.) y luego los atributos propios de un empleado (por ejemplo, `matricula`, `sector`, `sueldo`, etc.). Decimos entonces que la clase `Empleado` hereda de la clase `Persona` y (si `Persona` no hereda de ninguna otra clase) entonces `Persona` hereda de `Object`. Así, un empleado es una persona y una persona es un *object*; por lo tanto, por transitividad, un empleado también es un *object*.

Es muy importante saber que todas las clases heredan de la clase base `Object` ya que los métodos definidos en esta clase serán comunes a todas las demás (las que vienen con el lenguaje y las que programemos nosotros mismos).

En este momento nos interesa estudiar dos de los métodos que heredamos de `Object`: el método `toString` y el método `equals`.

14.2.3 El método `toString`

Todas las clases heredan de `Object` el método `toString`; por esto, podemos invocar este método sobre cualquier objeto de cualquier clase. Tal es así que cuando hacemos:

```
System.out.println(obj);
```

siendo `obj` un objeto de cualquier clase, lo que implícitamente estamos haciendo es:

```
System.out.println( obj.toString() );
```

ya que `System.out.println` invoca el método `toString` del objeto que recibe como parámetro. Es decir, `System.out.println` "sabe" que cualquiera sea el tipo de datos del objeto, este seguro tendrá el método `toString`.

Por este motivo, en la clase `Fecha` podemos sobrescribir el método `toString` para indicar el formato con el que queremos que se impriman las fechas.

Decimos que sobrescribirmos un método cuando programamos en una clase el mismo método que heredamos de nuestra clase base.



Decimos que sobrescribirmos un método cuando programamos en una clase el mismo método que heredamos de nuestra clase base.

A continuación, veremos cómo sobrescribir el método `toString` en la clase `Fecha`.

```
package libro.cap14.fechas;

public class Fecha
{
   private int dia;
   private int mes;
   private int anio;

   // sobrescribimos el metodo toString (lo heredamos de Object)
   public String toString()
   {
      // retorna una cadena tal como queremos que se vea la fecha
      return dia+"/"+mes+"/"+anio;
   }

   // :
   // setters y getters...
   // :
}
```

Ahora simplemente podremos imprimir el objeto `f` y la salida será la esperada: `"2/10/1970"` (considerando el ejemplo que analizamos más arriba).

14.2.4 El método `equals`

Otro de los métodos que heredamos de la clase `Object` y que se utiliza para comparar objetos es `equals`. El lector recordará que utilizamos este método para comparar cadenas. Pues bien, la clase `String` lo hereda de `Object` y lo sobrescribe de forma tal que permite determinar si una cadena es igual a otra comparando uno a uno sus caracteres.

En nuestro caso tendremos que sobrescribir el método `equals` para indicar si dos fechas son iguales o no.

```
package libro.cap14.fechas;

public class Fecha
{
   private int dia;
   private int mes;
   private int anio;

   // sobrescribimos el metodo equals que heredamos de Object
   public boolean equals(Object o)
   {
      Fecha otra = (Fecha)o;
      return (dia==otra.dia) && (mes==otra.mes) && (anio==otra.anio);
   }

   // :
   // setters y getters...
   // toString...
   // :
}
```

Como vemos, el método `equals` retorna `true` si "nuestro día" es igual al día de la otra fecha y "nuestro mes" es igual al mes de la otra fecha y "nuestro año" es igual al año de la otra fecha. Si no es así, entonces retorna `false`.

14.2.5 Declarar y "crear" objetos

Para utilizar un objeto, no basta con declarar una variable. Además, hay que "crearlo". En la siguiente línea, declaramos un objeto de tipo `Fecha`, pero no lo creamos.

```
// declaramos un objeto de tipo fecha
Fecha f;
```

El objeto `f` que declaramos arriba no está listo para ser usado porque no fue "creado" (instanciado). Esto lo haremos a continuación:

```
// creamos (instanciamos) el objeto f
f = new Fecha();
```



Las dos líneas de código anteriores podrían resumirse en una única línea en la que declaramos e instanciamos al objeto `f`:

```
// declaramos e instanciamos el objeto f de la clase Fecha.
Fecha f = new Fecha();
```

Los objetos son punteros. Al declarar un objeto, estamos declarando un puntero que, inicialmente, apuntará a una dirección de memoria nula (`null`). El objeto no se podrá utilizar mientras que no apunte a una dirección de memoria válida.

En el siguiente programa, declaramos un objeto e intentamos utilizarlo sin haberlo creado. Al ejecutarlo obtendremos un error de tipo `NullPointerException`.

Los objetos son punteros. Al declarar un objeto, estamos declarando un puntero que, inicialmente, apuntará a una dirección de memoria nula (`null`). El objeto no se podrá utilizar mientras que no apunte a una dirección de memoria válida.

```
package libro.cap14.fechas;

public class TestFecha2
{
   public static void main(String[] args)
   {
      // definimos el objeto f (pero no lo creamos)
      Fecha f;
      f.setDia(2);           // aqui arroja un error y finaliza el programa
      f.setMes(10);          // no se llega a ejecutar
      f.setAnio(1970);       // no se llega a ejecutar

      System.out.println(f); // no se llega a ejecutar
   }
}
```

■

La salida de este programa es la siguiente:

```
Exception in thread "main" java.lang.NullPointerException
    at libro.cap14.fechas.TestFecha2.main(TestFecha2.java:9)
```



Este mensaje de error debe interpretarse de la siguiente forma: "el programa finalizó arrojando una excepción de tipo `NullPointerException` en la línea 9 de la clase `TestFecha2`, dentro del método `main`". Vemos también que las líneas de código posteriores no llegaron a ejecutarse.

Más adelante, estudiaremos en detalle el tema de "excepciones". Por el momento, solo diremos que son errores que pueden ocurrir durante la ejecución de un programa Java.

Las excepciones son errores que pueden ocurrir durante la ejecución de un programa Java.



Toda clase tiene, al menos, un constructor. Podemos programarlo, explícitamente, o bien, aceptar el constructor que, por defecto, Java definirá por nosotros.
El constructor de una clase es un método que se llama exactamente igual que la clase y solo puede invocarse como argumento del operador `new` al momento de crear objetos de la clase.

14.2.6 El constructor

El constructor de una clase es un método "especial" a través del cual podemos crear los objetos de la clase.

Toda clase tiene, al menos, un constructor. Podemos programarlo, explícitamente, o bien, aceptar el constructor que, por defecto, Java definirá por nosotros.

El constructor se utiliza para crear los objetos de las clases.

```
// creamos un objeto a traves del constructor por default
Fecha f = new Fecha();
```

En esta línea de código, declaramos y creamos el objeto `f` utilizando el constructor `Fecha()`. Vemos también que el operador `new` recibe como argumento al constructor de la clase.

Es decir que el constructor de una clase es un método que se llama exactamente igual que la clase y solo puede invocarse como argumento del operador `new` al momento de crear objetos de la clase.

Hasta ahora no hemos declarado un constructor para la clase `Fecha;` por lo tanto, los objetos de esta clase solo se podrán crear mediante el uso del constructor "nulo" o "por defecto" que Java define automáticamente para este fin.

Sería práctico poder crear objetos de la clase `Fecha` pasándoles los valores del día, mes y año a través del constructor. Con esto evitaremos la necesidad de invocar a los *setters* de estos atributos.

Agregaremos entonces en la clase `Fecha` un constructor que reciba tres enteros (día, mes y año) y los asigne a sus atributos.

```
package libro.cap14.fechas;

public class Fecha
{
   private int dia;
   private int mes;
   private int anio;

   // constructor
   public Fecha(int d, int m, int a)
   {
      dia = d;
      mes = m;
      anio = a;
   }

   // :
   // setters y getters...
   // toString...
   // equals...
   // :
}
```

Como podemos ver, en el constructor, recibimos los tres valores y los asignamos a los atributos correspondientes. También podríamos haber invocado, dentro del código del constructor, a los *setters*. Por ejemplo, en lugar de: `dia = d`, hacer: `setDia(d)`.

Es importante tener en cuenta que al declarar, explícitamente, un constructor perdemos el constructor por defecto. Por lo tanto, ahora, el siguiente código no compilará:

```
// esto ahora no compila porque en la clase Fecha
// no existe un constructor que no reciba argumentos
Fecha f = new Fecha();
```



Al declarar explícitamente un constructor perdemos el constructor por defecto.

En cambio, podremos crear fechas especificando los valores de sus atributos.

```
// Ahora si... creamos la fecha del 2 de octubre de 1970
Fecha f = new Fecha(2, 10, 1970);
```

14.2.7 Repaso de lo visto hasta aquí

Antes de seguir incorporando conceptos sería conveniente hacer un pequeño repaso de todo lo expuesto hasta el momento.

- Toda clase hereda, directa o indirectamente, de la clase base `Object`.
- Los métodos de la clase `Object` son comunes a todas las clases.
- De `Object` siempre heredaremos los métodos `toString` y `equals`.
- Podemos sobrescribir estos métodos para definir el formato de impresión de los objetos de nuestras clases y el criterio de comparación respectivamente.
- "Sobrescribir" significa reescribir el cuerpo de un método que estamos heredando sin modificar su prototipo.
- Los objetos no pueden ser utilizados hasta tanto no hayan sido creados.
- Para crear objetos utilizamos el constructor de la clase.
- Todas las clases tienen, al menos, un constructor.
- Podemos programar nuestro constructor o utilizar el constructor por defecto.
- Al programar explícitamente un constructor entonces "perdemos" el constructor nulo o por defecto.

Ejemplo: Compara dos fechas ingresadas por el usuario.

En el siguiente programa, le pedimos al usuario que ingrese dos fechas y las comparamos utilizando su método `equals`.

```
package libro.cap14.fechas;

import java.util.Scanner;

public class TestFecha3
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      // el usuario ingresa los datos de la fecha
      System.out.print("Ingrese Fecha1 (dia, mes y anio): ");
      int dia = scanner.nextInt();
      int mes = scanner.nextInt();
      int anio = scanner.nextInt();

      // creamos un objeto de la clase Fecha
      Fecha f1 = new Fecha(dia,mes,anio);
```

```
        // el usuario ingresa los datos de la fecha
        System.out.print("Ingrese Fecha2 (dia, mes y anio): ");
        dia = scanner.nextInt();
        mes = scanner.nextInt();
        anio = scanner.nextInt();

        // creamos un objeto de la clase Fecha
        Fecha f2 = new Fecha(dia,mes,anio);

        System.out.println("Fecha 1 = "+f1);
        System.out.println("Fecha 2 = "+f2);

        if( f1.equals(f2) )
        {
           System.out.println("Son iguales!");
        }
        else
        {
           System.out.println("Son distintas...");
        }
   }
}
```

En este ejemplo creamos los objetos `f1` y `f2` de la clase `Fecha` utilizando el constructor que recibe tres `int` (el que programamos nosotros). Luego los mostramos con `System.out.println` que invocará sobre estos objetos el método `toString`. La salida será *dd/mm/aaaa* porque este es el formato de impresión que definimos al sobrescribir dicho método. Por último, comparamos las dos fechas utilizando el método `equals`, que también sobrescribimos.

14.2.8 Convenciones de nomenclatura

Antes de seguir avanzando considero conveniente explicar las convenciones que comúnmente son aceptadas y aplicadas a la hora de asignar nombres a las clases, métodos, atributos, constantes y variables.

14.2.8.1 Los nombres de las clases

Las clases siempre deben comenzar con mayúscula. En el caso de tener un nombre compuesto por más de una palabra, cada inicial también debe estar en mayúscula. Por ejemplo:

```
public class NombreDeLaClase
{
   // :
}
```

14.2.8.2 Los nombres de los métodos

Los métodos siempre deben comenzar en minúscula. En el caso de tener un nombre compuesto por más de una palabra, cada inicial debe comenzar en mayúscula salvo, obviamente, la primera.

```
public void nombreDelMetodo()
{
   //...
}
```

14.2.8.3 Los nombres de los atributros

Para los atributos se utiliza la misma convención que definimos para los métodos: comienzan en minúscula y, si el nombre consta de más de una palabra, entonces cada inicial, salvo la primera, debe ir en mayúscula.

```
public class Persona
{
   private String nombre;
   private Date fechaDeNacimiento;

   // :
}
```

14.2.8.4 Los nombres de las variables de instancia

Las variables de instancia que no sean consideradas como atributos pueden definirse a gusto del programador siempre y cuando no comiencen con mayúscula. Algunos programadores utilizan la "notación húngara", que consiste en definir prefijos que orienten sobre el tipo de datos de la variable. Así, si tenemos una variable `contador` de tipo `int`, según esta notación, la llamaremos **`iContador`** y si tenemos una variable `fin` de tipo `boolean` será **`bFin`**.

14.2.8.5 Los nombres de las constantes

Para las constantes se estila utilizar solo letras mayúsculas. Si el nombre de la constante está compuesto por más de una palabra, entonces debemos utilizar el "guión bajo" o *underscore* para separarlas.

```
public static final int NOMBRE_DE_LA_CONSTANTE = 1;
```

14.2.9 Sobrecarga de métodos

En el capítulo anterior, vimos que al método `indexOf` de la clase `String` le podemos pasar tanto un argumento de tipo `char` como uno de tipo `String`. Veamos:

```
String s = "Esto es una cadena";
int pos1 = s.indexOf("e"); // retorna 5
int pos2 = s.indexOf('e'); // retorna 5
```

En este código invocamos al método `indexOf` primero pasándole un argumento de tipo `String` y luego, uno de tipo `char`. Ambas invocaciones son correctas, funcionan bien y son posibles porque el método `indexOf` de la clase `String` está "sobrecargado". Es decir, el mismo método puede invocarse con diferentes tipos y/o cantidades de argumentos.

Decimos que un método está sobrecargado cuando admite recibir más de una combinación de tipos y/o cantidades de argumentos. Esto se logra escribiendo el método tantas veces como tantas combinaciones diferentes queremos que el método admita.



Un método está sobrecargado cuando admite más de una combinación de tipos y/o cantidades de argumentos.

A continuación, veremos cómo sobrecargar el constructor de la clase `Fecha` para poder crear fechas especificando sus atributos o bien sin especificar ningún valor.

```
package libro.cap14.fechas;

public class Fecha
{
   private int dia;
   private int mes;
   private int anio;
```

```
   // constructor recibe dia, mes y anio
   public Fecha(int d, int m, int a)
   {
      dia = d;
      mes = m;
      anio = a;
   }

   // constructor sin argumentos
   public Fecha()
   {
   }

   // :
   // setters y getters...
   // toString...
   // equals...
   // :
}
```

Luego, las siguientes líneas de código son correctas:

```
// creamos una fecha indicando los valores iniciales
Fecha f1 = new Fecha(2,10,1970);

// creamos una fecha sin indicar valores iniciales
Fecha f2 = new Fecha();
f2.setDia(4);      // asignamos el dia
f2.setMes(6);      // asignamos el mes
f2.setAnio(2008); // asignamos el anio
```

Es importante no confundir “sobrecarga” con “sobrescritura”:

- Sobrecargamos un método cuando lo programamos más de una vez, pero con diferentes tipos y/o cantidades de parámetros.
- Sobrescribimos un método cuando el método que estamos programando es el mismo que heredamos de su padre. En este caso, tenemos que respetar su encabezado (cantidades y tipos de parámetros y tipo del valor de retorno) ya que de lo contrario lo estaremos sobrecargando.

La propuesta ahora es hacer que la clase `Fecha` permita crear fechas a partir de una cadena de caracteres con este formato: “*dd/mm/aaaa*”. Para esto, tendremos que agregar (sobrecargar) un constructor que reciba como parámetro una cadena.

La estrategia será la siguiente: suponiendo que la cadena que recibimos como parámetro en el constructor es “15/06/1973”, entonces:

1. Ubicamos la posición de la primera ocurrencia de `'/'` (la llamaremos `pos1`).
2. Ubicamos la posición de la última ocurrencia de `'/'` (la llamaremos `pos2`).
3. Tomamos la subcadena ubicada entre 0 y `pos1` (no inclusive), la convertimos a `int` y la asignamos al atributo `dia`.
4. Tomamos la subcadena ubicada entre `pos1+1` y `pos2` (no inclusive), la convertimos a `int` y la asignamos en el atributo `mes`.
5. Tomamos la subcadena ubicada a partir de `pos2+1`, la convertimos a `int` y la asignamos en el atributo `anio`.

```java
package libro.cap14.fechas;

public class Fecha
{
   private int dia;
   private int mes;
   private int anio;

   public Fecha(String s)
   {
      // buscamos la primera ocurrencia de '/'
      int pos1 = s.indexOf('/');

      // buscamos la ultima ocurrencia de '/'
      int pos2 = s.lastIndexOf('/');

      // procesamos el dia
      String sDia = s.substring(0,pos1);
      dia = Integer.perseInt(sDia);

      // procesamos el mes
      String sMes = s.substring(pos1+1,pos2);
      mes = Integer.parseInt(sMes);

      // procesamos el anio
      String sAnio = s.substring(pos2+1);
      anio = Integer.parseInt(sAnio);
   }

   // :
   // otros constructores...
   // setters y getters...
   // toString...
   // equals...
   // :
}
```

Ahora podremos crear fechas a partir de una cadena con formato “*dd/mm/aaaa*” o bien especificando sus atributos por separado.

```java
// creamos una fecha a partir de los tres valores por separado
Fecha f1 = new Fecha(25, 10,2004);

// creamos una fecha a partir de una cadena con formato dd/mm/aaaa
Fecha f2 = new Fecha("25/10/2004");

// trabajamos con las fechas, no importa como fueron creadas
if( f1.equals(f2) )
{
   System.out.println("Las fechas son iguales!");
}
```



Uno de los objetivos que buscamos cuando programamos clases es encapsular la complejidad que emerge de las operaciones asociadas a sus atributos.

14.2.10 Encapsulamiento

Uno de los objetivos que buscamos cuando programamos clases es encapsular la complejidad que emerge de las operaciones asociadas a sus atributos. Esto significa que debemos facilitarle la vida al programador que utilice objetos de nuestras clases exponiendo las operaciones que pueda llegar a necesitar, pero ocultando su complejidad.

Por ejemplo, una operación aplicable a una fecha podría ser sumarle o restarle días. Si definimos el método `addDias` en la clase `Fecha`, entonces quien utilice esta clase podrá sumarle días a sus fechas sin tener que conocer el algoritmo que resuelve el problema.

```
// creamos una fecha
Fecha f = new Fecha("23/12/1980");

// le sumamos 180 dias
f.addDias(180);

// mostramos como quedo la fecha luego de sumarle estos dias
System.out.println(f);
```

Para facilitar los cálculos y no perder el tiempo en cuestiones que aquí resultarían ajenas consideraremos que todos los meses tienen 30 días. Por lo tanto, en esta versión de la clase `Fecha` los años tendrán 360 días (12 meses de 30 días cada uno, 12 x 30=360).

Con estas consideraciones, el algoritmo para sumar días a una fecha consistirá en convertir la fecha a días, sumarle los días que corresponda y asignar los nuevos valores del día, mes y año en los atributos.

Entonces serán tres los métodos que vamos a programar:

- El método `addDias` será el método que vamos a “exponer” para que los usuarios de la clase puedan invocar y así sumarle días a sus fechas.
- Desarrollaremos también el método `fechaToDias` que retornará un entero para representar la fecha expresada en días. Este método no lo vamos a “exponer”. Es decir, no será visible para el usuario: será `private` (privado).
- Por último, desarrollaremos el método inverso: `diasToFecha` que, dado un valor entero que representa una fecha expresada en días, lo separará y asignará los valores que correspondan a los atributos `dia`, `mes` y `anio`. Este método también será `private` ya que no nos interesa que el usuario lo pueda invocar.

```
package libro.cap14.fechas;

public class Fecha
{
   private int dia;
   private int mes;
   private int anio;

   // retorna la fecha expresada en dias
   private int fechaToDias()
   {
      // no requiere demasiada explicacion...
      return anio*360+mes*30+dia;
   }
```

```
   // asigna la fecha expresada en dias a los atributos
   private void diasToFecha(int i)
   {
      // dividimos por 360 y obtenemos el año
      anio = (int)i/360;

      // del resto o residuo de la division anterior
      // podremos obtener el mes y el dia
      int resto = i%360;

      // el mes es el resto dividido 30
      mes = (int) resto/30;


      // el resto de la division anterior son los dias
      dia = resto%30;

      // ajuste por si el dia quedo en cero
      if( dia == 0)
      {
         dia=30;
         mes--;
      }

      // ajuste por si el mes quedo en cero
      if( mes == 0)
      {
         mes=12;
         anio--;
      }
   }

   public void addDias(int d)
   {
      // convertimos la fecha a dias y le sumamos d
      int sum=fechaToDias()+d;

      // la fecha resultante (sum) se separa en dia, mes y año
      diasToFecha(sum);
   }

   // :
   // constructores...
   // setters y getters...
   // toString...
   // equals...
   // :
}
```
■

Ejemplo: Suma días a una fecha.

Ahora podemos desarrollar una aplicación en la que el usuario ingrese una fecha expresada en formato *dd/mm/aaaa* y una cantidad de días para sumarle, y nuestro programa le mostrará la fecha resultante.

```
package libro.cap14.fechas;

import java.util.Scanner;

public class TestFecha4
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      // el usuario ingresa los datos de la fecha
      System.out.print("Ingrese Fecha (dd/mm/aaaa): ");

      // leemos la fecha (cadena de caracteres) ingresada
      String sFecha = scanner.nextLine();

      // creamos un objeto de la clase Fecha
      Fecha f = new Fecha(sFecha);

      // lo mostramos
      System.out.println("La fecha ingresada es: "+f);

      // el usuario ingresa una cantidad de dias por sumar
      System.out.print("Ingrese dias a sumar (puede ser negativo): ");
      int diasSum = scanner.nextInt();

      // le sumamos esos dias a la fecha
      f.addDias(diasSum);

      // mostramos la nueva fecha (con los dias sumados)
      System.out.println("sumando "+diasSum+" dias, queda: "+f);
   }
}
```

14.2.11 Visibilidad de los métodos y los atributos

En el ejemplo anterior, hablamos de “exponer” y “ocultar”. Esto tiene que ver con el nivel de visibilidad que podemos definir para los métodos y los atributos.

Aquellos métodos y atributos declarados como `public` (públicos) serán visibles desde cualquier otra clase. Por el contrario, los métodos y atributos declarados como `private` (privados) estarán encapsulados y solo podrán ser invocados y manipulados dentro de la misma clase.

En el caso de la clase `Fecha`, los atributos `dia`, `mes` y `anio` son `private`; por lo tanto, no pueden ser manipulados desde el programa principal.

```
public class ProgramaPrincipal
{
   public static void main(String args)
   {
      Fecha f = new Fecha();

      // error de compilacion ya que el atributo es private
      f.dia = 21;
   }
}
```

Lo mismo pasa con los métodos `fechaToDias` y `diasToFecha`. Estos métodos son privados y no se pueden invocar desde afuera de la clase.

```
public class ProgramaPrincipal
{
   public static void main(String args)
   {
      Fecha f = new Fecha("25/2/1980");

      // el metodo fechaToDias es private, error de compilacion
      int dias = f.fechaToDias();
   }
}
```

En cambio, podremos invocar cualquier método `public` como pueden ser los constructores, los métodos `toString`, `equals`, `addDias`, etcétera. Si la clase tuviese atributos declarados `public` entonces podríamos manipularlos en cualquier método de cualquier otra clase.



Si para desarrollar un método complejo tenemos que dividirlo (estructurarlo) en varios métodos más simples, estos probablemente deban ser `private` para prevenir que el usuario los pueda invocar.

En general, se estila que los atributos sean `private` y los métodos `public`. Si para desarrollar un método complejo tenemos que dividirlo (estructurarlo) en varios métodos más simples, estos probablemente deban ser `private` para prevenir que el usuario los pueda invocar. Con esto, evitaremos confundir al programador que usa nuestras clases.

Existen otros dos niveles de visibilidad: `protected` y `friendly` pero los estudiaremos más adelante.

14.2.12 Packages (paquetes)

Los paquetes proporcionan un mecanismo que permite organizar las clases en función de un determinado criterio. Además, constituyen un *namespace* (espacio de nombres) que posibilita que varias clases tengan el mismo nombre siempre y cuando estén ubicadas en paquetes diferentes.

Para que una clase se ubique en un determinado paquete, la primera línea de código del archivo que la contiene debe ser `package`, seguida del nombre del paquete.

¿Qué hubiera sucedido si a nuestra clase `Fecha` la hubiésemos llamado `Date`? No sucedería absolutamente nada. Java trae por lo menos dos clases llamadas `Date`; la nuestra hubiera sido la tercera. Siendo así, ¿cómo pueden convivir tres clases con el mismo nombre? Es simple, están ubicadas en diferentes paquetes. Java provee las clases: `java.util.Date` y `java.sql.Date`. La nuestra, de haberla llamado `Date`, sería: `libro.cap14.fechas.Date`.

Físicamente, los paquetes son directorios o carpetas. Para ubicar una clase dentro del paquete `x`, el `.class` debe estar grabado en la carpeta `x` y el código fuente debe comenzar con la línea `package x`.

14.2.13 Estructura de paquetes y la variable `CLASSPATH`

Los proyectos Java deben estar ubicados dentro de una carpeta que será la base de las carpetas que constituyen los paquetes. A esta carpeta la llamaremos *package root* (raíz de los paquetes).

Por ejemplo, la clase `Fecha` está en el paquete `libro.cap14.fechas`. Si consideramos como *package root* a la carpeta `c:\misproyectos\demolibro`, entonces la estructura del proyecto será la siguiente:

```
c:\
 |_misproyectos\
       |_demolibro\
                |_src\
                |   |_libro\
                |       |_cap14\
                |           |_fechas\
                |                |_Fecha.java
                |                |_TestFecha.java
                |                |_     :
                |
                |_bin\
                    |_libro\
                        |_cap14\
                            |_fechas\
                                 |_Fecha.class
                                 |_TestFecha.class
                                 |_     :
```

Cuando trabajamos con Eclipse esto es totalmente "transparente" ya que la herramienta lo hace automáticamente. Sin embargo, si queremos ejecutar un programa Java desde la línea de comandos tendremos que tener en cuenta la estructura anterior y los pasos por seguir serán los siguientes:

1. Definir la variable de ambiente `CLASSPATH=c:\misproyectos\demolibro\bin`
2. Asegurarnos de que la carpeta que contiene `java.exe` esté dentro del `PATH`

Con esto, podremos ejecutar nuestro programa Java de la siguiente manera:

```
java libro.cap14.fechas.TestFecha
```

En la variable `CLASSPATH`, podemos definir más de una ruta. Esto permite que las clases que programamos en un proyecto puedan utilizar clases que fueron desarrolladas en otros proyectos. Las diferentes rutas deben separarse con `;` (punto y coma):

```
set CLASSPATH=c:\...\pkgroot1;c:\...\pkgroot2;...
```



Llamamos API al conjunto de paquetes (con sus clases y métodos) que están disponibles para utilizar en nuestros programas.
Una API es un paquete o un conjunto de paquetes cuyas clases son funcionalmente homogéneas y están a nuestra disposición

14.2.14 Las APIs (Application Programming Interface)

Dado que los paquetes agrupan clases funcionalmente homogéneas, es común decir que determinados paquetes constituyen una API.

Llamamos API al conjunto de paquetes (con sus clases y métodos) que están disponibles para utilizar en nuestros programas.

Todos los paquetes que provee Java constituyen la API de Java, pero podemos ser más específicos y, por ejemplo, decir que el paquete `java.net` constituye la API de *networking* y también podemos decir que el paquete `java.sql` contituye la API de acceso a bases de datos (o la API de JDBC).

En resumen, una API es un paquete o un conjunto de paquetes cuyas clases son funcionalmente homogéneas y están a nuestra disposición.

Las API suelen documentarse en páginas HTML con una herramienta que se provee como parte del JDK: el javadoc. En este punto, le recomiendo al lector mirar el videotutorial que explica cómo utilizar esta herramienta.

Es común utilizar APIs provistas por terceras partes o bien desarrollar las nuestras propias y ponerlas a disposición de terceros. Dado que una API puede contener cientos o miles de clases, Java provee una herramienta para unificar todos los archivos (todos los `.class`) en un único archivo con extensión `.jar`. Esta herramienta es parte del JDK y su uso está documentado en el videotutorial correspondiente.

14.2.15 Representación gráfica UML

Como dice el refrán, “una imagen vale más que mil palabras”. Un gráfico puede ayudarnos a ordenar los conceptos e ideas a medida que los vamos incorporando.

Si bien no es tema de este libro, utilizaremos algunos de los diagramas propuestos por UML (*Unified Modeling Language*) cuando considere que pueden aportar claridad.

UML (o “lenguaje unificado de modelado”) es, en la actualidad, el lenguaje gráfico de modelado de objetos más difundido y utilizado. Su gran variedad de diagramas facilita las tareas de análisis, diseño y documentación de sistemas.



UML (o “lenguaje unificado de modelado”) es, en la actualidad, el lenguaje gráfico de modelado de objetos más difundido y utilizado. Su gran variedad de diagramas facilita las tareas de análisis, diseño y documentación de sistemas.

En este caso utilizaremos un diagrama de clases y paquetes para representar, gráficamente, lo que estudiamos más arriba. Los paquetes se representan como “carpetas” y las clases se representan en “cajas”. Las flechas indican una relación de herencia entre clases. Si de una clase sale una flecha que apunta hacia otra clase será porque la primera es una subclase de la segunda.

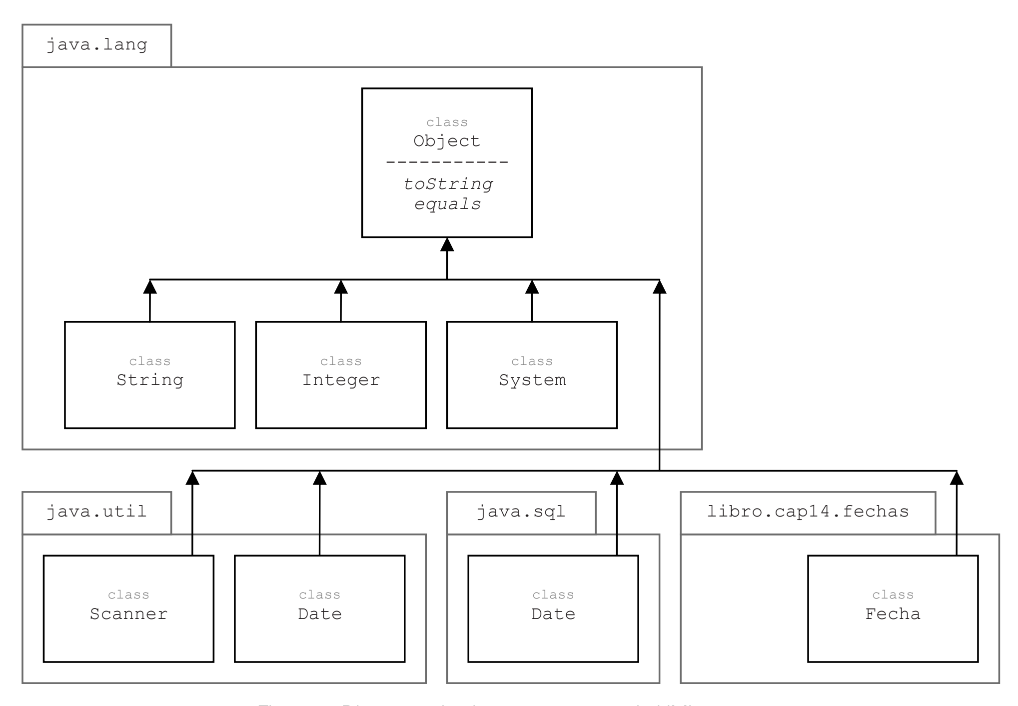


Fig. 14.1 Diagrama de clases y paquetes de UML.

El diagrama muestra algunas de las clases que hemos estado utilizando ubicándolas dentro del paquete al que pertenecen. También muestra la relación de herencia entre estas clases y la clase base `Object` en la que, además, se destacan los métodos `equals` y `toString`, indicando así que todas las clases los heredan.

Podemos observar que existen dos clases `Date`, pero esto no ocasiona ningún problema porque cada una está ubicada en un paquete diferente.

Obviamente, la clase `Object` tiene más métodos, los paquetes `java.lang`, `java.util` y `java.sql` tienen más clases y la API de Java provee muchos más paquetes. En el diagrama solo representamos aquellos elementos que nos interesa reflejar.



En el diagrama solo representamos aquellos elementos que nos interesa reflejar.

14.2.16 Importar clases de otros paquetes

Nuestras clases pueden utilizar otras clases, independientemente del paquete en el que estén contenidas; simplemente "las importamos" y las usamos. Para esto, se utiliza la sentencia `import`.

Si observamos bien, en todos los programas que hemos desarrollado utilizamos la sentencia `import java.util.Scanner`. Esto nos permitió utilizar la clase `Scanner` que, evidentemente, nosotros no desarrollamos.

Sin embargo, también utilizamos otras clases que no necesitamos importar. Por ejemplo: `String`, `System`, `Integer`, `Object`. Todas estas clases, y muchas más, están ubicadas en el paquete `java.lang`. Este paquete se importa solo; no es necesario importarlo explícitamente.

El lector, probablemente, relacionará la sentencia `import` con la instrucción de preprocesador `include` de C. Pues bien, no funcionan de la misma manera. La primera (el `import` de Java) solo define referencias, rutas en las cuales el compilador debe buscar las clases que utilizamos en nuestros programas. La segunda (el `include` de C) incluye, físicamente, el contenido de un archivo dentro de otro.

14.3 Herencia y polimorfismo

Como mencionamos anteriormente la herencia permite definir clases en función de otras clases ya existentes. Diremos que la "clase derivada" o la "subclase" hereda los métodos y los atributos de la "clase base". Esto posibilita redefinir el comportamiento de los métodos heredados y/o extender su funcionalidad.

En la sección anterior, trabajamos con la clase `Fecha`. Supongamos que no tenemos acceso al código de esta clase. Es decir, podemos utilizarla pero no la podemos modificar porque, por ejemplo, fue provista por terceras partes. Hagamos de cuenta que no la desarrollamos nosotros. De ese modo, supongamos que, aunque la clase `Fecha` nos resulta útil, funciona bien y es muy práctica, queremos modificar la forma en que una fecha se representa a sí misma cuando invocamos su método `toString`.

La solución será crear una nueva clase que herede de `Fecha` y que modifique la manera en que esta se representa como cadena. Esto lo podremos hacer sobrescribiendo el método `toString`. Llamaremos a la nueva clase `FechaDetallada` y haremos que se represente así: "25 de octubre de 2009".

```
package libro.cap14.misclases;

import libro.cap14.fechas.Fecha;

public class FechaDetallada extends Fecha
{
   private static String meses[]={"Enero"
                                  ,"Febrero"
                                  ,"Marzo"
                                  ,"Abril"
                                  ,"Mayo"
                                  ,"Junio"
                                  ,"Julio"
                                  ,"Agosto"
                                  ,"Septiembre"
                                  ,"Octubre"
                                  ,"Noviembre"
                                  ,"Diciembre"};
```

```
   // constructor nulo
   public FechaDetallada()
   {
   }

   // constructor que recibe dia, mes y anio
   public FechaDetallada(int d, int m, int a)
   {
      setDia(d);
      setMes(m);
      setAnio(a);
   }

   public String toString()
   {
      return getDia()+" de "+meses[getMes()-1]+" de "+getAnio();
   }
}
```

La clase `FechaDetallada` hereda de la clase base `Fecha` y sobrescribe el método `toString` para retornar una representación con más nivel de detalle que la que provee la implementación de su padre.

Para indicar que una clase hereda de otra, se utiliza la palabra `extends`. Decimos entonces que `FechaDetallada` "extiende" a `Fecha`. Otras expresiones válidas son:

- `FechaDetallada` "hereda" de `Fecha`
- `FechaDetallada` "es una especie" de `Fecha`
- `FechaDetallada` "es hija" de `Fecha`
- `FechaDetallada` "es una subclase" de `Fecha`
- `FechaDetallada` "subclasea" a `Fecha`

La lógica de programación que utilizamos para resolver el método `toString` en la clase `FechaDetallada` es simple: definimos un `String[]` que contenga los nombres de los meses, luego retornamos una cadena de caracteres concatenando el día, seguido del nombre del mes y el año.

Notemos que si la fecha corresponde al mes 10, por ejemplo, el nombre "octubre" se encuentra en la posición 9 del *array* porque en Java los *array* comienzan desde cero.

El *array* `meses` fue definido como `static`. Esto significa que es una "variable de clase", pero este tema lo estudiaremos más adelante.

Notemos también que para acceder a los atributos `dia`, `mes` y `anio` que la clase `FechaDetallada` hereda de `Fecha` fue necesario utilizar los *getters*. Esto se debe a que los atributos son privados y, si bien existen en la clase derivada, no son accesibles sino a través de sus métodos de acceso.

Lamentablemente, los constructores no se heredan. Por este motivo, los desarrollamos explícitamente ya que, de no hacerlo así, el único constructor disponible sería el contructor nulo o por defecto.

Probemos la clase `FechaDetallada` con el siguiente programa donde creamos un objeto tipo `FechaDetallada`, le asignamos valor a sus atributos y lo imprimimos.

```
package libro.cap14.misclases;

public class TestFechaDetallada
{
   public static void main(String[] args)
   {
      FechaDetallada f = new FechaDetallada();
      f.setDia(25);
      f.setMes(10);
      f.setAnio(2009);

      System.out.println(f);
   }
}
```

En el siguiente diagrama, reflejamos las relaciones que existen entre los objetos del programa anterior.

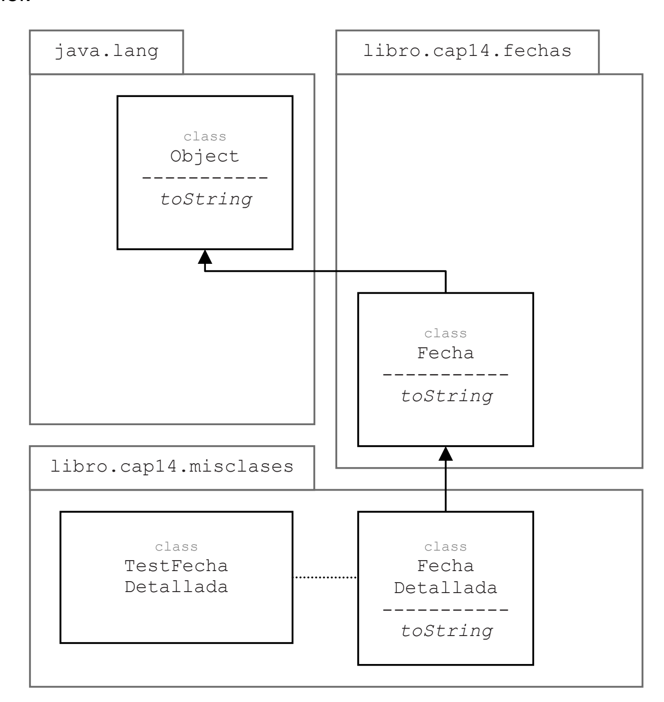


Fig. 14.2 Representación de las clases `Fecha` y `FechaDetallada`.

El diagrama representa la relación de herencia que existe entre las clases `FechaDetallada`, `Fecha` y `Object`, cada una en su propio paquete. También refleja que las clases `Fecha` y `FechaDetallada` sobrescriben el método `toString` que heredan de `Object`. Por último, muestra que la clase `TestFechaDetallada` "usa" la clase `FechaDetallada`.

Obviamente, la clase `TestFechaDetallada` también hereda de `Object`. Sin embargo, esta relación no es relevante en nuestro ejemplo, por lo que decidimos no reflejarla.

14.3.1 Polimorfismo

Los objetos nunca dejan de reconocerse como miembros de una determinada clase. Por tal motivo, independientemente de cual sea el tipo de datos de la variable que los contenga, ante la invocación de cualquiera de sus métodos siempre reaccionan como su propia clase lo define.



Los objetos nunca dejan de reconocerse como miembros de una determinada clase. Ante la invocación de cualquiera de sus métodos siempre reaccionan como su propia clase lo define.

Este concepto resultará más claro sobre un ejemplo concreto: en el siguiente programa asignamos una instancia tipo `FechaDetallada` a una variable tipo `Fecha` y otra instancia, también tipo `FechaDetallada` a una variable tipo `Object`. Luego imprimimos los dos objetos con `System.out.println`, el cual, como ya sabemos, invocará al método `toString`.

```
package libro.cap14;

import libro.cap14.fechas.Fecha;
import libro.cap14.misclases.FechaDetallada;

public class TestPolimorfismo
{
   public static void main(String[] args)
   {
      // a fec (de tipo Fecha) le asignamos un objeto FechaDetallada
      Fecha fec = new FechaDetallada(25,02,2009);

      // a obj (de tipo Object) le asignamos un objeto FechaDetallada
      Object obj = new FechaDetallada(3,12,2008);

      // Imprimimos los dos objetos
      System.out.println("fec = "+fec); // invoca a toString de fec
      System.out.println("obj = "+obj); // invoca a toString de obj
   }
}
```

Podemos asignar una instancia de la clase `FechaDetallada` en una variable tipo `Fecha` porque, dada la relación de herencia, una `FechaDetallada` "es una especie de" `Fecha`.

Análogamente, como `Fecha` hereda de `Object`, resulta que una `FechaDetallada` también "es una especie de" `Object`, por lo que una variable tipo `Object` puede contener una instancia de `FechaDetallada`.

Luego, la salida del programa será:

```
fec = 25 de Febrero de 2009
obj = 3 de Diciembre de 2008
```

Aunque el tipo de datos de `fec` es `Fecha` y el tipo de datos de `obj` es `Object`, ambos objetos se representan a sí mismos con el formato que define la clase `FechaDetallada` porque los dos contienen referencias a instancias de esta clase.

Como comentamos más arriba, un objeto nunca se olvida de su clase. Así, *por polimorfismo*, se invocará al método `toString` definido en la clase a la que este pertenece. (Recordemos que `System.out.println(x)` imprime la cadena que retorna el método `toString` del objeto `x`).

Para terminar de comprender la idea pensemos en una clase con un método que permita imprimir un conjunto de objetos. Este método recibirá un `Object[]` (léase "object array") para recorrerlo y mostrar en la consola la representación de cada uno de sus elementos.

```
package libro.cap14;

public class MuestraConjunto
{
   public static void mostrar(Object[] arr)
   {
      for( int i=0; i<arr.length; i++ )
      {
         System.out.println("arr["+i+"] = "+ arr[i]);
      }
   }
}
```

Como podemos ver, dentro del método `mostrar` no conocemos el tipo de datos de los objetos que contiene el *array*. Como `arr` es un `Object[]` y todas las clases heredan de `Object`, entonces `arr[i]` puede ser un objeto de cualquier tipo.

Sin embargo, esto no nos impide imprimirlos ya que cada objeto `arr[i]` acudirá a su propia clase cuando `System.out.println` invoque a su método `toString`.

Veamos ahora el programa principal donde invocamos al método `mostrar` de la clase `MuestraConjunto`.

```
package libro.cap14;

import libro.cap14.fechas.Fecha;
import libro.cap14.misclases.FechaDetallada;

public class TestMuestraConjunto
{
   public static void main(String[] args)
   {
      Object[] arr = { new Fecha(2,10,1970)
                     , new FechaDetallada(23,12,1948)
                     , new String("Esto es una cadena")
                     , new Integer(34) };

      // como el metodo es estatico lo invocamos a traves de la clase
      MuestraConjunto.mostrar(arr);
   }
}
```

Efectivamente, `arr` contiene objetos de tipos de datos diferentes: `Fecha`, `FechaDetallada`, `String` e `Integer`. Aun así, cuando se les invoca el método `toString` cada uno reacciona como su propia clase lo define.

La salida de este programa será la siguiente:

```
arr[0] = 2/10/1970
arr[1] = 23 de Diciembre de 1948
arr[2] = Esto es una cadena
arr[3] = 34
```

Polimorfismo es la característica fundamental de la programación orientada a objetos. Este tema será profundizado a lo largo del presente capítulo. Por el momento basta con esta breve explicación.

Antes de terminar debemos mencionar que el método `mostrar` de la clase `MuestraConjunto` es estático. Esto lo convierte en un "método de la clase". Si bien este tema lo analizaremos más adelante, podemos ver que en el método `main` lo invocamos directamente sobre la clase haciendo `MuestraConjunto.mostrar`. Es el mismo caso que el de `Integer.parseInt` que utilizamos para convertir cadenas de caracteres a números enteros.

14.3.2 Constructores de subclases

Lo primero que hace el constructor de una clase derivada es invocar al constructor de su clase base. Si esto no se define, explícitamente, (programándolo) entonces se invocará al constructor nulo de la clase base y si este no existe, tendremos un error de compilación.

En la clase `FechaDetallada`, programamos los siguientes constructores:

```
public FechaDetallada()
{
}

public FechaDetallada(int d,int m,int a)
{
   setDia(d);
   setMes(m);
   setAnio(a);
}
```

Como en ninguno hacemos referencia explícita al constructor de la clase base, implícitamente, cuando invoquemos a cualquiera de estos constructores, Java invocará al constructor nulo de `Fecha`.

En este punto le recomiendo al lector comentar, momentáneamente, el código del constructor nulo de la clase `Fecha` y volver a compilar la clase `FechaDetallada`. Así, obtendrá el siguiente error de compilación:

```
Implicit super constructor Fecha() is undefined
         for default constructor. FechaDetallada.java (line 6)
```

Este mensaje indica que no está definido el constructor nulo (o sin argumentos) en la clase `Fecha`.

Observemos que en el mensaje de error aparece la palabra "super". El significado y uso de esta palabra reservada lo estudiaremos a continuación.

14.3.3 La referencia `super`

Si en la clase base tenemos un constructor que recibe una determinada combinación de parámetros, en las clases derivadas tendremos que programarlo, explícitamente, o bien, resignarnos a no tenerlo porque, como comentamos más arriba, los constructores no se heredan.

Como en la clase `Fecha` definimos tres constructores, lo razonable será programar estos mismos constructores también en la clase `FechaDetallada`.

```
package libro.cap14.misclases;

public class FechaDetallada extends Fecha
{
   // :
   // definicion del array meses...
   // :

   public FechaDetallada(int dia, int mes, int anio)
   {
      // invocamos al constructor del padre
      super(dia,mes,anio);
   }

   public FechaDetallada(String s)
   {
      // invocamos al constructor del padre
      super(s);
   }

   public FechaDetallada()
   {
      // invocamos al constructor del padre
      super();
   }

   // :
   // metodo toString...
   // :
}
```

Con esto, en la clase `FechaDetallada`, tendremos los mismos constructores que ofrece la clase `Fecha`.

En cada uno de estos constructores, invocamos al constructor del padre a través de la palabra `super`. Esta palabra representa al constructor del padre que, en este caso, es el constructor de la clase `Fecha`.

Ahora bien, ¿qué ocurrirá si en alguno de estos constructores no invocamos, explícitamente, al constructor del padre? Sucederá lo siguiente:

1. Por omisión se invocará al constructor nulo de la clase `Fecha`, pero como este existe no tendremos ningún error de compilación.
2. Como el constructor nulo de `Fecha` no hace nada, quedarán sin asignar los atributos `dia`, `mes` y `anio`.
3. Cuando imprimamos una fecha con `System.out.println` se invocará al método `toString` de `FechaDetallada`, que accederá al *array* `meses[mes-1]`, pero como el atributo `mes` no tiene valor se generará un error en tiempo de ejecución, un `ArrayIndexOutOfBoundException`.

Le sugiero al lector tratar de modificar el constructor que recibe tres enteros anulando la llamada al constructor del padre de la siguiente manera:

```
public FechaDetallada(int dia, int mes, int anio)
{
   // super(dia,mes,anio);
}
```

Luego, si hacemos:

```
public static void main(String args[])
{
   FechaDetallada f = new FechaDetallada(25,10,2009);
   Sytstem.out.println(f);
}
```

tendremos el siguiente error en tiempo de ejecución:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
at libro.cap14....FechaDetallada.toString(FechaDetallada.java:42)
   at java.lang.String.valueOf(String.java:2577)
   at java.io.PrintStream.print(PrintStream.java:616)
   at java.io.PrintStream.println(PrintStream.java:753)
   at lib....TestFechaDetallada.main(TestFechaDetallada.java:15)
```

Como vemos, `super` se utiliza para hacer referencia al constructor del padre. En Java no existe la “herencia múltiple”; por lo tanto, cualquiera sea la clase, su padre siempre será único.



En Java no existe la “herencia múltiple”; por lo tanto, cualquiera sea la clase, su padre siempre será único.

Ahora bien, como la clase base puede tener varios constructores entonces, dependiendo de los argumentos que le pasemos a `super`, se invocará al constructor que concuerde, exactamente, con esa combinación de parámetros.

La palabra `super` también puede utilizarse como un objeto y, en este caso, funciona como una “referencia al padre”.

Para ejemplificar su uso, extenderemos todavía más la funcionalidad de nuestras clases desarrollando la clase `FechaHora` como una subclase de `FechaDetallada`. La clase `FechaHora` permitirá también representar una hora (hora, minutos y segundos).

```
package libro.cap14.misclases;

public class FechaHora extends FechaDetallada
{
   private int hora;
   private int minuto;
   private int segundo;

   public FechaHora(String sFecha, int hora, int min, int seg)
   {
      super(sFecha);
      this.hora = hora;
      this.minuto = min;
      this.segundo = seg;
   }

   public String toString()
   {
      // invocamos al metodo toString de nuestro padre
      return super.toString()+" ("+hora+":"+minuto+":"+segundo+")";
   }

   // :
   // otros constructores...
   // setters y getters...
   // :
}
```

■

Esta clase extiende la funcionalidad de `FechaDetallada` proveyendo, además, la capacidad de almacenar la hora.

En el método `toString` (que sobrescribimos) utilizamos la palabra `super` para invocar al método `toString` de `FechaDetallada`. A la cadena que obtenemos le concatenamos otra cadena representando la hora, y eso es lo que retornamos.

Ahora veremos un programa donde utilizamos un objeto tipo `FechaHora` y lo imprimimos.

```
package libro.cap14.misclases;

public class TestFechaHora
{
   public static void main(String[] args)
   {
      FechaHora fh = new FechaHora("25/2/2006",14,30,10);
      System.out.println(fh);
   }
}
```

La salida de este programa será:

```
25 de Febrero de 2006 (14:30:10)
```

14.3.4 La referencia `this`

Así como `super` hace referencia al constructor del padre, la palabra `this` hace referencia a los otros constructores dentro de una misma clase.

Para ejemplificar el uso de `this` replantearemos el desarrollo de los constructores de la clase `FechaDetallada` de la siguiente manera:

```
package libro.cap14.misclases;

public class FechaDetallada extends Fecha
{
   // :
   // definicion del array meses...
   // :

   public FechaDetallada(int dia, int mes, int anio)
   {
      super(dia,mes,anio);
   }

   public FechaDetallada()
   {
      // invocamos al constructor de tres int pasando ceros
      this(0,0,0);
   }

   // :
   // contructor que recibe un String
   // metodo toString...
   // :
}
```

Como vemos, desde el constructor que no recibe parámetros invocamos al constructor que recibe tres enteros y le pasamos valores cero. Con esto, daremos valores iniciales (aunque absurdos) a la fecha que está siendo creada.

También, `this` puede ser usado como referencia a “nosotros mismos”. Esto ya lo utilizamos en la clase `Fecha` cuando en los *setters* hacíamos:

```
public void setDia(int dia)
{
   this.dia = dia;
}
```

Dentro de este método asignamos el valor del parámetro `dia` al atributo `dia`. Con `this.dia` nos referimos al atributo `dia` (que es un miembro o variable de instancia de la clase) y lo diferenciamos del parámetro `dia`, que simplemente es una variable automática del método.

Los conceptos de “instancia”, “atributo” y “variable de instancia” los estudiaremos en detalle más adelante, en este mismo capítulo.



Una clase abstracta es una clase que tiene métodos que no pueden ser desarrollados por falta de información concreta. Estos métodos se llaman “métodos abstractos” y deben desarrollarse en las subclases, cuando esta información esté disponible.

14.3.5 Clases abstractas

En ocasiones reconocemos la existencia de objetos que, claramente, son elementos de una misma clase y, sin embargo, sus operaciones se realizan de manera muy diferente. El caso típico para estudiar este tema es el de las figuras geométricas.

Nadie dudaría en afirmar que “toda figura geométrica describe un área cuyo valor se puede calcular”. Sin embargo, para calcular el área de una figura geométrica será necesario conocer cuál es esa figura. Es decir: no basta con saber que se trata de una figura geométrica; necesitamos también conocer de qué figura estamos hablando.

Dicho de otro modo, podemos calcular el área de un rectángulo, podemos calcular el área de un círculo y podemos también calcular el área de un triángulo pero no podemos calcular el área de una figura geométrica si no conocemos concretamente cual es esa figura.

Una clase abstracta es una clase que tiene métodos que no pueden ser desarrollados por falta de información concreta. Estos métodos se llaman “métodos abstractos” y deben desarrollarse en las subclases, cuando esta información esté disponible.

En nuestro ejemplo, `FiguraGeometrica` será una clase abstracta con un único método abstracto: el método `area`.

```
package libro.cap14.figuras;

public abstract class FiguraGeometrica
{
   // metodo abstracto
   public abstract double area();

   public String toString()
   {
      return "area = " + area();
   }
}
```

Las clases abstractas se definen como `abstract class`. El método `area` es un método abstracto, lo definimos como `abstract` y lo dejamos sin resolver finalizando su declaración con `;` (punto y coma).



Las clases abstractas no pueden ser instanciadas porque hay métodos que aún no han sido implementados.

Las clases abstractas no pueden ser instanciadas. Es decir, no podemos crear objetos de clases declaradas como abstractas porque, por definición, hay métodos que aún no han sido implementados.

No podemos hacer esto:

```
// MAL, esto no compila
FiguraGeometrica fg = new FiguraGeometrica();
```

Cabe plantear el siguiente interrogante para el lector: en la clase `FiguraGeometrica` sobrescribimos el método `toString` y dentro de este invocamos al método `area` (que es abstracto); ¿esto tiene sentido?

En general, las clases abstractas deben ser "subclaseadas". Toda clase que herede de una clase abstracta tiene que sobrescribir los métodos abstractos de su padre o bien deberá ser declarada abstracta y, por lo tanto, no se podrá instanciar.

Ahora podemos pensar en figuras geométricas tales como el círculo, el rectángulo y el triángulo. Serán subclases de `FiguraGeometrica` en las cuales tendremos la información suficiente como para sobrescribir el método `area` ya que estaremos hablando de figuras concretas. Por ejemplo, el área de un círculo se calcula como *π x radio*^{2} (léase "pi por radio al cuadrado") y el de un rectángulo será *base x altura* mientras que el de un triángulo será *base x altura / 2*. Es decir, el área se calcula en función de la figura geométrica y de sus atributos.

Figura	Atributos	Área
Círculo	*radio*	*π x radio*^{2}
Rectángulo	*base, altura*	*base x altura*
Triángulo	*base, altura*	*base x altura / 2*

Fig. 14.3 Figuras geométricas, atributos y fórmulas para calcular el área.

```
package libro.cap14.figuras;

public class Rectangulo extends FiguraGeometrica
{
   private double base;
   private double altura;

   public Rectangulo(double b, double h)
   {
      base = b;
      altura = h;
   }

   public double area()
   {
      return base * altura;
   }

   // :
   // setters y getters
   // :
}
```

Como vemos, en la clase `Rectangulo`, definimos los atributos que caracterizan a dicha figura y sobrescribimos, adecuadamente, el método `area` retornando el producto de sus atributos `base` y `altura`. Lo mismo haremos con las clases `Circulo` y `Triangulo`.

```
package libro.cap14.figuras;

public class Circulo extends FiguraGeometrica
{
   private int radio;

   public Circulo(int r)
   {
      radio = r;
   }

   public double area()
   {
      // retornamos "PI por radio al cuadrado"
      return Math.PI*Math.pow(radio,2);
   }
}
```

El número π está definido como una constante estática en la clase `Math`. Para acceder a su valor, lo hacemos a través de `Math.PI`. Para calcular la potencia *radio*^{2} utilizamos el método `pow`, también estático y definido en la misma clase `Math`.
El concepto de "estático" lo estudiaremos más adelante pero, como comentamos anteriormente, sabemos que los métodos estáticos pueden invocarse directamente sobre la clase sin necesidad de instanciarla; es el caso de `Math.pow`, `System.out.println`, `Integer.parseInt`, etcétera.

```
package libro.cap14.figuras;

public class Triangulo extends FiguraGeometrica
{
   private double base;
   private double altura;

   public Triangulo(int b, int h)
   {
      base = b;
      altura = h;
   }

   public double area()
   {
      return base*altura/2;
   }
}
```

Pensemos ahora en un programa que utilice estas clases:

```
package libro.cap14.figuras;

public class TestFiguras
{
   public static void main(String[] args)
   {
      Circulo c = new Circulo(4);
      Rectangulo r = new Rectangulo(10,5);
      Triangulo t = new Triangulo(3,6);
```

```
      System.out.println(c);
      System.out.println(r);
      System.out.println(t);
   }
}
```

La salida será:

```
area = 50.26548245743669
area = 50.0
area = 9.0
```

Este resultado demuestra lo siguiente:

1. Las clases `Circulo`, `Rectangulo` y `Triangulo` heredan de `FiguraGeometrica` el método `toString`. Recordemos que dentro de este método invocábamos al método abstracto `area`.
2. Cuando en `toString` invocamos al método `area` en realidad estamos invocando a la implementación concreta del método `area` del objeto sobre el cual se invocó a `toString`. Por este motivo, resulta que el cálculo del área es correcto para las tres figuras que utilizamos en el `main`.

14.3.6 Constructores de clases abstractas

Que una clase abstracta no pueda ser instanciada no significa que no pueda tener constructores.

¿Qué sentido tiene declarar un constructor en una clase que no podremos instanciar? El sentido es el de "obligar" a las subclases a *settear* los valores de los atributos de la clase base.

Agregaremos el atributo `nombre` en clase `FiguraGeometrica` y también un constructor que permita especificar su valor. Así, cada figura podrá guardar su nombre y proveer información más específica cuando invoquemos a su método `toString`.

```
package libro.cap14.figuras;

public abstract class FiguraGeometrica
{
   private String nombre;

   // metodo abstracto
   public abstract double area();

   // agregamos un constructor
   public FiguraGeometrica(String nom)
   {
      nombre = nom;
   }

   // ahora en el toString mostramos tambien el nombre
   public String toString()
   {
      return nombre +" (area = "+ area()+") ";
   }
```

```
   public String getNombre()
   {
      return nombre;
   }

   public void setNombre(String nombre)
   {
      this.nombre = nombre;
   }
}
```

Ahora tenemos que modificar las subclases e invocar explícitamente al constructor definido en la clase base.

```
package libro.cap14.figuras;

public class Rectangulo extends FiguraGeometrica
{
   private double base;
   private double altura;

   public Rectangulo(double b, double h)
   {
      super("Rectangulo"); // constructor del padre
      base = b;
      altura = h;
   }

   // :
   // metodo area
   // setters y getters...
   // :
}
```

Lo primero que debemos hacer en el constructor de la clase derivada es invocar al constructor del padre. Como el constructor de `FiguraGeometrica` espera recibir el nombre de la figura, le pasamos como argumento “nuestro” propio nombre que, en este caso, es “Rectangulo”.

Algún lector con experiencia podrá pensar que estamos “hardcodeando” el nombre “Rectangulo”, pero no es así. Se trata del nombre de la figura y este nunca podrá cambiar arbitrariamente.

Apliquemos los cambios en `Circulo` y `Triangulo`.

```
package libro.cap14.figuras;

public class Circulo extends FiguraGeometrica
{
   private int radio;

   public Circulo(int r)
   {
      super("Circulo");
      radio = r;
   }
```

```
    // :
    // metodo area
    // setters y getters...
    // :
}
```

```
package libro.cap14.figuras;

public class Triangulo extends FiguraGeometrica
{
   private double base;
   private double altura;

   public Triangulo(int b, int h)
   {
      super("Triangulo");
      base = b;
      altura = h;
   }

   // :
   // metodo area
   // setters y getters...
   // :
}
```

Ahora al ejecutar el programa principal obtendremos la siguiente salida:

```
Circulo (area = 50.26548245743669)
Rectangulo (area = 50.0)
Triangulo (area = 9.0)
```

Por último, informalmente, dijimos que los métodos estáticos pueden invocarse directamente sobre las clases sin tener que instanciarlas. Podríamos definir un método estático en la clase `FiguraGeometrica` que permita calcular el área promedio de un conjunto de figuras.

```
package libro.cap14.figuras;

public abstract class FiguraGeometrica
{
   private String nombre;

   // metodo abstracto
   public abstract double area();

   public static double areaPromedio(FiguraGeometrica arr[])
   {
      int sum=0;
      for( int i=0; i<arr.length; i++ )
      {
         sum += arr[i].area();
      }
```

```
      return sum/arr.length;
   }

   // :
   // constructor
   // setters y getters...
   // :
}
```

Es fundamental notar la importancia y el poder de abstracción que logramos al combinar métodos abstractos y polimorfismo.

En el método `areaPromedio`, recorremos el conjunto de figuras y sobre cada elemento `arr[i]` invocamos al método `area` sin preocuparmos por conocer cuál es la figura concreta.

El simple hecho de no poder instanciar a `FiguraGeometrica` nos garantiza que `arr[i]` únicamente podrá tener objetos de alguna de las implementaciones concretas de esta clase. Luego, por polimorfismo, `arr[i].area()` llamará al método `area` de la clase concreta, nunca al de la clase abstracta porque esta no puede ser instanciada.

Veamos un programa donde calculamos el área promedio de las figuras geométricas contenidas en un *array*.

```
package libro.cap14.figuras;

public class TestAreaPromedio
{
   public static void main(String[] args)
   {
      FiguraGeometrica arr[] = { new Circulo(23)
                               , new Rectangulo(12,4)
                               , new Triangulo(2,5) };

      double prom = FiguraGeometrica.areaPromedio(arr);

      System.out.println("Promedio = " + prom);
   }
}
```

14.3.7 Instancias

Los objetos son instancias de las clases porque cada uno (cada objeto) mantiene diferentes combinaciones de valores en sus atributos. Las expresiones “crear un objeto” e “instanciar la clase” muchas veces son sinónimos, aunque no siempre “instanciar” implica “crear un objeto” en el sentido de definir una variable para que lo contenga.

Comenzaremos analizando una clase muy simple:

```
package libro.cap14.instancias;

public class X
{
   private int a;
   private int b;
```

```
   public X(int a, int b)
   {
      this.a = a;
      this.b = b;
   }

   public String toString()
   {
      return "("+a+","+b+")";
   }

   // :
   // setters y getters...
   // :
}
```

Definimos una clase `X` que tiene los atributos `a` y `b`, un constructor y el método `toString`. Analicemos ahora un programa que utilice esta clase.

```
package libro.cap14.instancias;

public class TestX
{
   public static void main(String[] args)
   {
      X x1 = new X(5,4);
      X x2 = new X(2,7);

      System.out.println("x1 = " + x1);
      System.out.println("x2 = " + x2);
   }
}
```

En este programa `x1` y `x2` son dos objetos de la clase `X`. También es correcto decir que `x1` y `x2` son instancias de `X`. Las dos expresiones son equivalentes.

Luego de ejecutar el programa anterior obtendremos el siguiente resultado:

```
x1 = (5,4)
x2 = (2,7)
```

Esto demuestra que cada objeto (o cada instancia) mantiene valores propios para sus atributos. Es decir, el atributo `a` de `x1` tiene el valor 5 pero el atributo `a` de `x2` tiene el valor 2. El atributo `b` de `x1` vale 4 mientras que el valor del atributo `b` de `x2` es 7.

Como estudiamos anteriormente, todos los objetos de la clase `X` tendrán un atributo `a` y un atributo `b`, pero cada objeto podrá mantener valores independientes y particulares para estos atributos.

14.3.8 Variables de instancia

Siguiendo con el ejemplo anterior, decimos que `a` y `b` son variables de instancia de la clase `X` ya que cada instancia de `X` tendrá sus propios valores para estas variables.

Los atributos de las clases son variables de instancia pero las variables de instancia no siempre serán consideradas como atributos. Para comprender esto analizaremos la clase `Persona` cuyo código es el siguiente:

```
package libro.cap14.instancias;

import libro.cap14.fechas.Fecha;
import libro.cap14.misclases.FechaDetallada;

public class Persona
{
   private String nombre;          // atributo
   private String dni;             // atributo
   private Fecha fechaNacimiento;  // atributo

   private int cont = 0;           // variable de estado

   public Persona(String nombre, String dni, Fecha fecNac)
   {
      this.nombre = nombre;
      this.dni = dni;
      this.fechaNacimiento = fecNac;
   }

   public String toString()
   {
      cont++;

      return nombre + ", DNI: " + dni
                    + ", nacido el: " + fechaNacimiento
                    + " (" + cont + ")";
   }

   // :
   // setters y getters...
   // :
}
```

La clase `Persona` tiene cuatro variables de instancia: `nombre`, `dni`, `fechaNacimiento` y `cont`; sin embargo, solo las primeras tres pueden ser consideradas como atributos.

La variable de instancia `cont` es utilizada para contar cuántas veces se invoca al método `toString` sobre el objeto. Por esto, dentro del método `toString` lo primero que hacemos es incrementar su valor.

Las variables `nombre`, `dni` y `fechaNacimiento` tienen que ver con "la persona", pero la variable `cont` simplemente es un recurso de programación: un contador.

Como podemos ver, el hecho de que una variable de instancia sea considerada como atributo es subjetivo y quedará a criterio del programador.

En el siguiente programa, creamos dos objetos de la clase `Persona` y los imprimimos varias veces:



Los atributos de las clases son variables de instancia pero las variables de instancia no siempre serán consideradas como atributos.

```
package libro.cap14.instancias;

import libro.cap14.misclases.FechaDetallada;

public class TestPersona
{
   public static void main(String[] args)
   {
      Persona p1 = new Persona("Juan"
                              ,"21773823"
                              ,new FechaDetallada(23,3,1971));

      Persona p2 = new Persona("Pablo"
                              ,"19234452"
                              ,new FechaDetallada(12,6,1968));

      System.out.println(p1);
      System.out.println(p1);
      System.out.println(p1);
      System.out.println("---");
      System.out.println(p2);
      System.out.println(p2);
      System.out.println("---");
      System.out.println(p1);
      System.out.println(p1);
   }
}
```

La salida del programa es:

```
Juan, DNI: 21773823, nacido el: 23 de Marzo de 1971 (1)
Juan, DNI: 21773823, nacido el: 23 de Marzo de 1971 (2)
Juan, DNI: 21773823, nacido el: 23 de Marzo de 1971 (3)
---
Pablo, DNI: 19234452, nacido el: 12 de Junio de 1968 (1)
Pablo, DNI: 19234452, nacido el: 12 de Junio de 1968 (2)
---
Juan, DNI: 21773823, nacido el: 23 de Marzo de 1971 (4)
Juan, DNI: 21773823, nacido el: 23 de Marzo de 1971 (5)
```

Esto demuestra que cada instancia mantiene los valores de sus variables. El objeto `p1` (Juan) lo imprimimos tres veces y luego dos veces más. Su variable `cont` llegó a 5 mientras que al objeto `p2` (Pablo) lo imprimimos dos veces, con lo que su variable `cont` llegó a contar hasta 2.

Por último, cuando instanciamos a `p1` (y también a `p2`) hicimos lo siguiente:

```
Persona p1 = new Persona("Juan"
                        ,"21773823"
                        ,new FechaDetallada(23,3,1971));
```

El tercer argumento es una instancia de `FechaDetallada`. Esta instancia no la asignamos a ningún objeto (a ninguna variable); por lo tanto, dentro del método `main` no la podremos volver a utilizar.

14.3.9 Variables de la clase

Así como en las variables de instancia cada objeto puede mantener valores propios e independientes respecto de los otros objetos de la clase, existe la posibilidad de definir variables al nivel de la clase. Estas variables son comunes a todas las instancias y, por lo tanto, son compartidas por todos sus objetos.



Las variables de la clase (estáticas) son comunes a todas las instancias. Por esto, son compartidas entre todos los objetos de la clase.

Volvamos a analizar el código de la clase `FechaDetallada`. En esta clase definimos un *array* con los nombres de los meses. Es evidente que los nombres de los meses son comunes a todas las fechas que podamos crear.

Dicho de otro modo, cualquiera sea la fecha que vayamos a representar con un objeto de la clase `FechaDetallada`, esta corresponderá a alguno de los 12 meses definidos en el *array*. Por este motivo, declaramos al *array* `meses` como una variable de la clase aplicándole el modificador `static`. Así todos los objetos utilizarán el mismo y único *array* de meses.

14.3.10 El garbage collector (recolector de residuos)

Cada vez que creamos un objeto estamos asignando memoria dinámicamente. El objeto funciona como un puntero que apunta a esta memoria dinámica. En lenguajes como C o Pascal, luego de asignar memoria dinámica tenemos que liberarla para que otros procesos la puedan utilizar. Por esto, los lenguajes de programación proveen funciones tales como `free` de C, `dispose` de Pascal o el operador `delete` de C++.

En Java esto no es responsabilidad del programador. Dentro de la máquina virtual existe un proceso que se ocupa de buscar y liberar la memoria que dejamos desreferenciada.

En el siguiente código, creamos dos instancias de la clase `X` analizada más arriba, pero al crear la segunda instancia dejamos desreferenciada la primera.

```
// definimos una variable de tipo X (la variable p)
X p;

// instanciamos a X y asignamos la instancia en p
p = new X(2,1);

// instanciamos a X y asignamos la instancia en p dejando
// desreferenciada a la instancia anterior
p = new X(5,4);
```

En el ejemplo comenzamos asignando a `p` la instancia `new X(2,1)`. Esta instancia puede ser accedida a través del objeto `p` para (por ejemplo) invocar sus métodos de acceso, su `toString`, etcétera.

Luego asignamos al objeto `p` una nueva instancia: `new X(5,4)` haciendo que `p` apunte a esta última y perdiendo así la referencia a la primera.

En casos como estos entra en juego el *garbage collector* (proceso recolector de residuos) que identifica a las instancias desreferenciadas y las elimina, liberando la memoria.

14.3.11 El método `finalize`



Antes de destruir a una instancia desreferenciada el *garbage collector* invoca al método `finalize` y luego la destruye.

Todas las clases heredan de `Object` un método llamado `finalize`.

Antes de destruir a una instancia desreferenciada el *garbage collector* invoca al método `finalize` y luego la destruye. Esto quiere decir que si sobrescribimos al método `finalize` podemos "hacer algo" antes de que la instancia pase a mejor vida.

Le propongo al lector analizar el siguiente programa para deducir lo que hace.

```
package libro.cap14.estaticas;

public class TestGC
{
   private static int cont = 0;

   public TestGC()
   {
      cont++;
      System.out.println(cont);
   }

   public void finalize()
   {
      cont--;
   }

   public static void main(String args[])
   {
      while( true )
      {
         new TestGC();
      }
   }
}
```

En esta clase definimos la variable `cont` estática (variable de clase) cuyo valor incrementamos en el constructor y decrementamos en el método `finalize`. Es decir, la incrementamos cuando se crean los objetos de la clase y la decrementamos cuando se destruyen. Como la variable es `static`, entonces será única y compartida por todos los objetos de la clase; por lo tanto, `cont` cuenta la cantidad de instancias activas de la clase `TestGC`.

14.3.12 Constantes

Es común declarar a las constantes como "variables de clase finales". Ejemplos de esto son, en la clase `java.lang.Math`, las constantes `PI` (3.1415...) y `E` (2.7182...).

```
public class Math
{
   public static final double PI = 3.1415;
   public static final double E  = 2.7182;
}
```

Al estar definidas como `public` (accesibles desde cualquier otra clase) y `static` (directamente sobre la clase) pueden utilizarse en cualquier clase o programa de la siguiente manera:

```
System.out.println("el numero PI es: " + Math.PI);
System.out.println("el numero E  es: " + Math.E);
```

14.3.13 Métodos de la clase

Los métodos definidos como estáticos (`static`) se convierten en métodos de la clase y pueden ser invocados, directamente, a través de la clase sin tener que instanciarla.

En los ejemplos anteriores, utilizamos varios métodos de clase provistos por Java como ser: `Math.pow` e `Integer.parseInt`. En general, podemos identificar estos métodos como "aquellos métodos cuyo valor de retorno está determinado, exclusivamente, en función de sus argumentos y, obviamente, no necesitan acceder a ninguna variable de instancia".

Pensemos en una clase `Numero` con un método `sumar` que reciba dos parámetros, `a` y `b`, y retorne su suma.

```
package libro.cap14.estaticas;

public class Numero
{
   public static double sumar(double a, double b)
   {
      return a+b;
   }
}
```

Evidentemente, el valor de retorno del método `sumar` depende exclusivamente de los valores de sus parámetros. Se trata entonces de un método de la clase. Desde cualquier clase o programa podemos invocar al método `sumar` haciendo:

```
// suma 2+5
double c = Numero.sumar(2,5);
```

Agreguemos ahora en la clase `Numero` una variable de instancia que permita guardar un valor concreto para cada objeto de la clase y un constructor a través del cual asignaremos su valor inicial. También sobrescribiremos el método `toString`.

```
package libro.cap14.estaticas;

public class Numero
{
   private double valor;

   public Numero(double v)
   {
      valor = v;
   }

   public String toString()
   {
      return Double.toString(valor);
   }

   public static double sumar(double a, double b)
   {
      return a+b;
   }

   // :
   // setters y getters...
   // :
}
```

Podemos pensar en sobrecargar al método `sumar` de forma tal que reciba un único parámetro y sume su valor de la variable de instancia. Claramente, este método será un método de instancia (no estático) ya que "tocará" a la variable de instancia `valor`.

```
package libro.cap14.estaticas;

public class Numero
{
   private double valor;

   public Numero sumar(double a)
   {
      valor+=a;
      return this;
   }

   public static double sumar(double a, double b)
   {
      return a+b;
   }

   // :
   // constructor
   // toString
   // setters y getters...
   // :
}
```

La versión sobrecargada del método `sumar` suma el valor del parámetro `a` al de la variable de instancia `valor` y retorna una referencia a la misma instancia (`this`) con la cual se está trabajando. Esto permitirá aplicar invocaciones sucesivas sobre el método `sumar` como veremos en el siguiente ejemplo.

```
package libro.cap14.estaticas;

public class TestNumero
{
   public static void main(String[] args)
   {
      // sumamos utilizando el metodo estatico
      double d = Numero.sumar(2,3);
      System.out.println(d);

      // definimos un numero con valor 5 y luego
      // sumamos 4 con el metodo sumar de instancia
      Numero n1 = new Numero(5);
      n1.sumar(4);

      System.out.println(n1);

      // sumamos concatenando invocaciones al metodo sumar
      n1.sumar(4).sumar(6).sumar(8).sumar(1);
      System.out.println(n1);
   }
}
```

14.3.14 Clases utilitarias

Se llama así a las clases que agrupan métodos estáticos para proveer cierto tipo de funcionalidad.

La clase `Math` (por ejemplo) es una clase utilitaria ya que a través de esta podemos invocar funciones trigonométricas, calcular logaritmos, realizar operaciones de potenciación, acceder a constantes numéricas como `PI` y `E`, etcétera.

Las clases que "wrappean" a los tipos primitivos también lo son. Por ejemplo, a través de la clase `Integer` podemos convertir una cadena en un valor entero y viceversa. También podemos realizar conversiones de bases numéricas, etcétera.

14.3.15 Referencias estáticas

Desde los métodos estáticos de una clase no tenemos acceso a variables o a métodos que no lo sean. En el siguiente ejemplo, definimos una variable de instancia `a` y un método, también de instancia, `unMetodo`. Luego intentamos accederlos desde el método `main` (que es estático), lo que genera errores de compilación.

```
package libro.cap14.estaticas;

public class TestEstatico
{
   private int a = 0;

   public void unMetodo()
   {
      System.out.println("este es unMetodo()");
   }

   public static void main(String[] args)
   {
      // no tenemos acceso a la variable a
      System.out.println("a vale " + a); // ERROR... NO COMPILA

      // no tenemos acceso al metodo unMetodo
      unMetodo();  // ERROR... NO COMPILA
   }
}
```

Los errores de compilación que obtendremos serán:

```
Cannot make a static reference to the non-static field a
  TestContextoEstatico.java, line 16
Cannot make a static reference to the non-static
  method unMetodo() from the type TestContextoEstatico
  TestContextoEstatico.java, line 19
```

Para solucionarlos podemos declarar la variable `a` y el método `unMetodo` como estáticos. Otra opción será, en el `main`, crear una instancia para acceder a la variable y al método como veremos a continuación:

```java
package libro.cap14.estaticas;

public class TestEstatico
{
   private int a = 0;

   public void unMetodo()
   {
      System.out.println("este es unMetodo()");
   }

   public static void main(String[] args)
   {
      // creamos la instancia t
      TestEstatico t = new TestEstatico();

      // accedemos a la variable a de la instancia t
      System.out.println("a vale " + t.a);

      // accedemos al metodo unMetodo() de la instancia t
      t.unMetodo();
   }
}
```

Dentro del método `main` podemos acceder a la variable `a` del objeto `t` sin utilizar su *getter* (que de hecho no existe). Esto es posible porque `main` es un método miembro de `TestEstatico`. Si `main` estuviese en otra clase, entonces la única forma de acceder a la variable `a` del objeto `t` sería a través de sus métodos de acceso.

14.3.16 Colecciones (primera parte)

En algunos de los ejemplos anteriores, desarrollamos métodos que trabajaban con conjuntos de objetos y en todos los casos utilizamos *arrays* para mantenerlos en memoria.

En esta sección veremos cómo desarrollar una clase que permita trabajar más amigablemente con conjuntos o colecciones de objetos. Con “más amigable” me refiero a que la clase debe proveer la funcionalidad para agregar, insertar, obtener, eliminar y buscar elementos dentro del conjunto. Además, la cantidad de objetos que pueda contener el conjunto debe ser “ilimitada”; es decir, que el usuario (programador que la utilice) pueda agregar elementos ilimitadamente siempre que disponga de la memoria necesaria.

Resumiendo, desarrollaremos una clase que tendrá los siguientes métodos:

```java
// agrega un elemento al final de la coleccion
public void agregar(Object elm);

// inserta un elemento en la i-esima posicion del array
public void insertar(Object elm, int i);

// retorna el i-esimo elemento de la coleccion
public Object obtener(int i);
```

```
// elimina y retorna el objeto en la i-esima posicion
public Object eliminar(int i);

// busca la primera ocurrencia del objeto especificado y retorna
// la posicion donde lo encuentra o un valor negativo si el
// objeto no se encontro
public int buscar(Object elm);

// retorna la cantidad de elementos del conjunto
public int cantidad();
```

Observemos que los métodos `insertar` y `agregar` reciben un `Object` como parámetro. Con esto, permitiremos agregar objetos de cualquier tipo a la colección. Análogamente, los métodos `obtener` y `eliminar` retornan un `Object` ya que, cualquiera sea el tipo de datos de un objeto, este siempre será `Object`.

Para mantener la colección de objetos en memoria, utilizaremos un *array* cuyo tamaño inicial podemos definir arbitrariamente o bien dejar que el usuario lo especifique pasándolo como argumento en el constructor.

La estrategia que utilizaremos para que la colección admita una cantidad ilimitada de objetos será la siguiente: agregaremos los objetos al *array* mientras este tenga espacio disponible. Cuando se llene, crearemos un nuevo *array* con el doble de la capacidad del anterior y trasladaremos los elementos del viejo *array* al nuevo. Al viejo *array* le asignaremos `null` para desreferenciarlo de forma tal que el *garbage collector* lo pueda liberar.

Por lo tanto, la clase `MiColeccion` tendrá dos variables de instancia: un `Object[]` para mantener la colección de objetos y un `int` que indicará la cantidad de elementos que actualmente tiene la colección, es decir, cuánta de la capacidad del *array* efectivamente está siendo utilizada.

```
package libro.cap14.colecciones;

public class MiColeccion
{
   private Object datos[] = null;
   private int len = 0;

   // en el constructor se especifica la capacidad inicial
   public MiColeccion(int capacidadInicial)
   {
      datos = new Object[capacidadInicial];
   }

   // sigue...
   // :
```

En el fragmento de código anterior, definimos las variables de instancia `datos` y `len` y un constructor a través del cual el usuario debe especificar la capacidad inicial que le quiera dar al *array*.

Veamos ahora los métodos `obtener` y `cantidad`. En estos métodos simplemente tenemos que retornar el objeto contenido en `datos[i]` y el valor de la variable `len` respectivamente.

```
// :
// viene de mas arriba...

// retorna el i-esimo elemento de la coleccion
public Object obtener(int i)
{
   return datos[i];
}

// indica cuantos elementos tiene la coleccion
public int cantidad()
{
   return len;
}

// sigue...
// :
```

Ahora analizaremos el método `insertar` cuyo objetivo es insertar un elemento en la *i-ésima* posición del *array*.
En este método verificamos si la capacidad del *array* está colmada. Si es así, entonces creamos un nuevo *array*, con el doble de la capacidad del anterior, y le copiamos los elementos de la colección. Luego le asignamos `null` para que el *garbage collector* libere la memoria que ocupa.
A continuación, desplazamos los elementos del *array* entre la última y la *i-ésima* posición para poder asignar en `datos[i]` el elemento que se pretende insertar. Al final, incrementamos el valor de la variable `len`.

```
// :
// viene de mas arriba...

public void insertar(Object elm, int i)
{
   if( len==datos.length )
   {
      Object aux[] = datos;
      datos = new Object[datos.length*2];

      for(int j=0; j<len; j++)
      {
         datos[j] = aux[j];
      }

      aux = null;
   }

   for( int j=len-1; j>=i; j-- )
   {
      datos[j+1]=datos[j];
   }

   datos[i] = elm;
   len++;
}

// sigue...
// :
```

Veamos ahora el método `buscar` que realiza una búsqueda secuencial sobre los elementos del *array* y retorna la posición donde encontró el elemento buscado o un valor negativo si no lo encontró.

```
// :
// viene de mas arriba...

public int buscar(Object elm)
{
   int i=0;

   // mientras no pasemos del tope y mientras no encuentre...
   for( ;i<len && !datos[i].equals(elm); i++ );

   // si no pasamos entonces encontramos; sino, no encontramos
   return i<len ? i : -1;
}

// sigue...
// :
```

El método `agregar`, cuyo objetivo es agregar un elemento al final del *array*, se resuelve fácilmente invocando al método `insertar` para insertar el elemento en la posición `len`.

```
// :
// viene de mas arriba...

public void agregar(Object elm)
{
   insertar(elm,len);
}

// sigue...
// :
```

Por último, veremos el código del método `eliminar` que elimina el *i-ésimo* elemento del *array* desplazando hacia arriba los elementos ubicados a partir de la posición *i*+1. Luego decrementa el valor de la variable `len` y retorna el elemento que ha sido eliminado de la colección.

```
   // :
   // viene de mas arriba...

   // elimina un elemento desplazando los demas hacia arriba
   public Object eliminar(int i)
   {
      Object aux = datos[i];

      for( int j=i; j<len-1; j++ )
      {
         datos[j] = datos[j+1];
      }

      len--;

      return aux;
   }
}
```

Ejemplo: Muestra una lista de nombres en el orden inverso al que fueron ingresados.

En el siguiente programa, le pedimos al usuario que ingrese nombres de personas. Cuando finaliza el ingreso de datos mostramos los nombres ingresados en orden inverso al original y además, por cada nombre, mostramos la cantidad de letras que tiene.

```
package libro.cap14.colecciones;

import java.util.Scanner;

public class TestMiColeccion
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      // creamos una coleccion con capacidad inicial = 5
      MiColeccion mc = new MiColeccion(5);

      // leemos el primer nombre
      System.out.println("Ingrese Nombre: ");
      String nom = scanner.next();

      while( !nom.equals("FIN") )
      {
         // insertamos siempre en la posicion 0
         mc.insertar(nom,0);

         // leemos el siguiente nombre
         nom = scanner.next();
      }

      String  aux;

      // recorremos la coleccion y tomamos cada uno de sus elementos
      for(int i=0; i<mc.cantidad(); i++ )
      {
         // el metodo obtener retorna un Object
         // entonces tenemos que "castear" (convertir) a String
         aux = (String) mc.obtener(i);

         System.out.println(aux +" - "+aux.length()+" caracteres");
      }
   }
}
```

La clase `MiColeccion` permite mantener en memoria una colección "ilimitada" de objetos de cualquier tipo porque es una colección de objetos de tipo `Object`. Esto tiene dos problemas: el primer problema es que, dada una instancia de `MiColeccion`, no podemos saber a priori el tipo de datos de sus objetos. Sabemos que son `Object` pero no sabemos concretamente de qué tipo son. Si no, analicemos los siguientes métodos:

```
// retorna una coleccion de personas
public MiColeccion obtenerPersonas(){...}

// retorna una coleccion de nombres
public MiColeccion obtenerNombres(){...}

// retorna una coleccion de numeros ganadores
public MiColeccion obtenerNumerosGanadores(){...}
```

Todos estos métodos retornan una instancia de `MiColeccion`, pero si no contamos con información adicional no podremos saber cuál es el tipo de datos de los objetos contenidos en la colección que retornan.

Por ejemplo, el método `obtenerNumerosGanadores` podría retornar una colección de `Integer` o una colección de `Long` o una colección de instancias de una clase `Jugada` que agrupe el número ganador, el nombre de la lotería y la fecha de la jugada.

El segundo problema está relacionado con el primero y surge cuando necesitamos acceder a alguno de los objetos de la colección. Como los métodos retornan `Object` tenemos que *castear* (convertir) al tipo de datos real del objeto y si no contamos con la información correspondiente no podremos saber a qué tipo de datos tenemos que *castear*.

En el programa que muestra los nombres y la cantidad de caracteres de cada nombre tomamos el *i-ésimo* objeto de la colección, lo "casteamos" a `String` y le invocamos el método `length`.

```
String  aux;
for(int i=0; i<mc.cantidad(); i++ )
{
   // "casteamos" a String
   aux = (String) mc.obtener(i);

   // como aux es String le invocamos el metodo length
   System.out.println(aux +" - "+aux.length()+" caracteres");
}
```

En este programa "sabemos" que la colección contiene *strings* porque nosotros mismos los asignamos más arriba, pero este no siempre será el caso.

Estos problemas los resolveremos haciendo que la clase `MiColeccion` sea una clase genérica.

14.3.17 Clases genéricas

Las clases genéricas permiten *parametrizar* los tipos de datos de los parámetros y valores de retorno de los métodos.

En el caso de la clase `MiColeccion`, podemos hacerla "genérica en `T`" de la siguiente manera:

```
public class MiColeccion<T>
{
    // :
   public void insertar(T elm, int i){ ... }
   public T obtener(int i) { ... }
    // :
}
```

La clase recibe el parámetro `T` que especifica el tipo de datos del parámetro `elm` del método `insertar` y el tipo de datos del valor de retorno del método `obtener`.

A continuación, veremos la versión genérica de la clase `MiColeccion`.

```
package libro.cap14.colecciones;

public class MiColeccion<T>
{
   public void agregar(T elm)
   {
      insertar(elm,len);
   }

   public void insertar(T elm, int i)
   {
      if( len==datos.length )
      {
         Object aux[] = datos;
         datos = new Object[datos.length*2];

         for(int j=0; j<len; j++)
         {
            datos[j] = aux[j];
         }

         aux = null;
      }

      for( int j=len-1; j>=i; j-- )
      {
         datos[j+1] = datos[j];
      }

      datos[i] = elm;
      len++;
   }

   public int buscar(T elm)
   {
      int i = 0;
      for( ;i<len && !datos[i].equals(elm); i++ );
      return i<len?i:-1;
   }

   public T eliminar(int i)
   {
      Object aux = datos[i];
      for( int j=i; j<len-1; j++ )
      {
         datos[j] = datos[j+1];
      }

      len--;
      return (T)aux;
   }

   public T obtener(int i)
   {
      return (T)datos[i];
   }
}
```

Para instanciar la clase genérica `MiColeccion` debemos especificar el tipo de datos que tomará el parámetro `T`. Lo hacemos de la siguiente manera:

```
// una coleccion de String con capacidad inicial de 5 elementos
MiColeccion<String> m1 = new MiColeccion<String>(5);

// una coleccion de Integer con capacidad inicial de 5 elementos
MiColeccion<Integer> m1 = new MiColeccion<Integer>(5);
```

Ahora podemos ver una nueva versión del programa que muestra, en orden inverso, un conjunto de nombres y muestra también cuántos caracteres tienen.

```
package libro.cap14.colecciones;

import java.util.Scanner;

public class TestMiColeccion
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Ingrese Nombre: ");
      String nom = scanner.next();

      // instanciamos una MiColeccion "especializada" en String
      MiColeccion<String> mc = new MiColeccion<String>(5);

      while( !nom.equals("FIN") )
      {
         mc.insertar(nom,0);
         nom = scanner.next();
      }

      String aux;
      for(int i=0; i<mc.cantidad(); i++ )
      {
         // no es necesario convertir porque el metodo obtener
         // retorna un String
         aux = mc.obtener(i);

         System.out.println(aux +" - "+aux.length()+" caracteres");
      }
   }
}
```

Ahora el método `obtener` de la clase `MiColeccion<String>` retorna un `String`, por lo que ya no es necesario *castear*. Por otro lado, el método `insertar` recibe un `String`. Si el lector hace el intento de insertar un objeto de otro tipo obtendrá un error de compilación ya que las clases genéricas proveen una validación de tipos de datos en tiempo de compilación.

Veamos ahora cómo quedarían los métodos `obtenerPersonas`, `obtenerNombres`, `obtenerNumerosGanadores`.



Las clases genéricas proveen una validación de tipos de datos en tiempo de compilación.

```
// retorna una coleccion de personas
public MiColeccion<Persona> obtenerPersonas(){...}

// retorna una coleccion de nombres
public MiColeccion<String> obtenerNombres(){...}

// retorna una coleccion de numeros ganadores
public MiColeccion<Jugada> obtenerNumerosGanadores(){...}
```

14.4 Interfaces

Como dijimos más arriba, en Java no existe la herencia múltiple, por lo que cada clase tiene un único padre. Esto de ninguna manera debe considerarse como una limitación ya que en un diseño de clases y objetos bien planteado una clase nunca debería necesitar heredar métodos y/o atributos de más de una única clase base.

Sin embargo, como Java es un lenguaje fuertemente "tipado", los objetos se manipulan a través de variables cuyos tipos de datos deben ser declarados con anticipación y esto, en ocasiones, puede limitar el diseño y la programación de nuestras aplicaciones.

Para comprender lo anterior formularemos la siguiente pregunta: ¿Qué tienen en común un teléfono celular, un telégrafo y una paloma mensajera? La respuesta es que los tres permiten enviar mensajes.

Si lo planteamos en términos de clases entonces deberíamos pensar en una clase base `Comunicador` con un método abstracto `enviarMensaje` y las clases `TelefonoCelular`, `PalomaMensajera` y `Telegrafo` heredando de `Comunicador`.

El hecho de que `TelefonoCelular` herede de `Comunicador` limita seriamente su funcionalidad ya que este, probablemente, debería heredar de la clase base `Telefono`. Análogamente, la clase `PalomaMensajera` debería heredar de `Paloma` y la clase `Telégrafo` podría heredar de `Reliquia`.



Las clases, abstractas o no se heredan. En cambio las *interfaces* se implementan.

Las *interfaces* proveen una solución a este tipo de problemas y constituyen uno de los recursos fundamentales para el diseño de aplicaciones Java.

En principio diremos lo siguiente: "una *interface* es una clase abstracta con todos sus métodos abstractos". Sin embargo, esto no es exactamente así ya que existe una diferencia fundamental entre una *interface* y una clase abstracta: las clases (abstractas o no) se "heredan" mientras que las *interfaces* se "implementan".

Por ejemplo, la clase `X` puede heredar de la clase base `Y` e implementar las *interfaces* `Z`, `T` y `W`. Claro que una clase que implementa una o más *interfaces* hereda todos sus métodos abstractos y debe sobrescribirlos adecuadamente ya que, de no hacerlo, quedará como una clase abstracta.

Con esto, volviendo a los elementos de comunicación, podríamos replantear el ejemplo de la siguiente manera: primero las clases base: `Telefono`, `Paloma` y `Reliquia`.

```
public class Telefono
{
   // atributos y metodos...
}

public class Paloma extends Ave
{
   // atributos y metodos...
}

public class Reliquia
{
   // atributos y metodos...
}
```

Ahora una *interface* `Comunicador` con su método `enviarMensaje`.

```
public interface Comunicador
{
   public void enviarMensaje(String mensaje);
}
```

Por último, las clases `TelefonoCelular`, `PalomaMensajera` y `Telegrafo`. Cada una extiende a una clase base diferente pero todas implementan la *interface* `Comunicador`; por lo tanto, todas heredan y sobrescriben el método abstracto `enviarMensaje`.

```
public class TelefonoCelular extends Telefono implements Comunicador
{
   public void enviarMensaje(String mensaje)
   {
      // hacer lo que corresponda aqui...
   }
}

public class PalomaMensajera extends Paloma implements Comunicador
{
   public void enviarMensaje(String mensaje)
   {
      // hacer lo que corresponda aqui...
   }
}

public class Telegrafo extends Reliquia implements Comunicador
{
   public void enviarMensaje(String mensaje)
   {
      // hacer lo que corresponda aqui...
   }
}
```

Ahora los objetos teléfono celular, paloma mensajera y telégrafo tienen una base en común: todos son `Comunicador` y, por lo tanto, pueden ser asignados en variables de este tipo de datos:

```
Comunicador t1 = new TelefonoCelular();
Comunicador t2 = new PalomaMensajera();
Comunicador t3 = new Telegrafo();
```

Claro que a los objetos `t1`, `t2` y `t3` únicamente se les podrá invocar el método `enviarMensaje` ya que este es el único método definido en la *interface* `Comunicador` (el tipo de datos de estos objetos).

A continuación, podremos apreciar la importancia de todo esto.

14.4.1 Desacoplamiento de clases

Supongamos que tenemos una clase utilitaria llamada `ComunicadorManager` con un método estático: `crearComunicador`:

```
public class ComunicadorManager
{
   public static Comunicador crearComunicador()
   {
      // una "paloma mensajera" es un "comunicador"
      return new PalomaMensajera();
   }
}
```

Utilizando esta clase podríamos escribir un programa como el que sigue:

```
public class MiAplicacionDeMensajes
{
   public static void main(String args[])
   {
      Comunicador c = ComunicadorManager.crearComunicador();
      c.enviarMensaje("Hola, este es mi mensaje");
   }
}
```

En este programa utilizamos la clase `ComunicadorManager` para obtener "un comunicador" a través del cual enviar nuestro mensaje. Lo interesante de esto es que en el método `main` no "hardcodeamos" ninguna de las clases que implementan la *interface* `Comunicador`; simplemente creamos un objeto comunicador utilizando el método `crearComunicador` e invocamos a su método `enviarMensaje`.

Dicho de otro modo, en el método `main` no sabemos con qué elemento comunicador estamos trabajando. Le pedimos a `ComunicadorManager` un objeto comunicador y lo obtuvimos pero no conocemos —y no nos debería interesar conocer— cuál es la clase concreta del objeto que obtuvimos y qué utilizaremos para enviar nuestro mensaje.

Ahora bien, evidentemente enviar un mensaje a través de un teléfono celular debe ser mucho más eficiente que enviarlo a través de una paloma mensajera. ¿Qué sucedería si modificamos el método `crearComunicador` de la clase `ComunicadorManager` y en lugar de retornar una instancia de `PalomaMensajera` retornamos una instancia de `TelefonoCelular`? ¿Qué cambios tendremos que hacer en el método `main`?

```
public class ComunicadorManager
{
   public static Comunicador obtenerComunicador()
   {
       // return new PalomaMensajera();
       // ahora retornamos un telefono celular
       // cuya clase tambien implementa Comunicador
      return new TelefonoCelular();
   }
}
```

La respuesta es: ninguno. En el método `main` trabajamos con un objeto comunicador y nos desentendemos de la necesidad de conocer qué tipo de comunicador es. No necesitamos saber si este objeto es una paloma mensajera, un teléfono celular o un telégrafo ya que todos estos objetos son comunicadores porque sus clases implementan la *interface* `Comunicador`.

Nuestro programa quedó totalmente desacoplado de la implementación puntual que utilizamos para enviar el mensaje. El cambio “de tecnología” que implica pasar de una paloma mensajera a un teléfono celular no tuvo ningún impacto negativo en nuestro programa (el método `main`); no fue necesario adaptarlo ni reprogramarlo.

Para tener una visión más global de las clases que intervienen en este ejemplo, analizaremos su diagrama de clases.

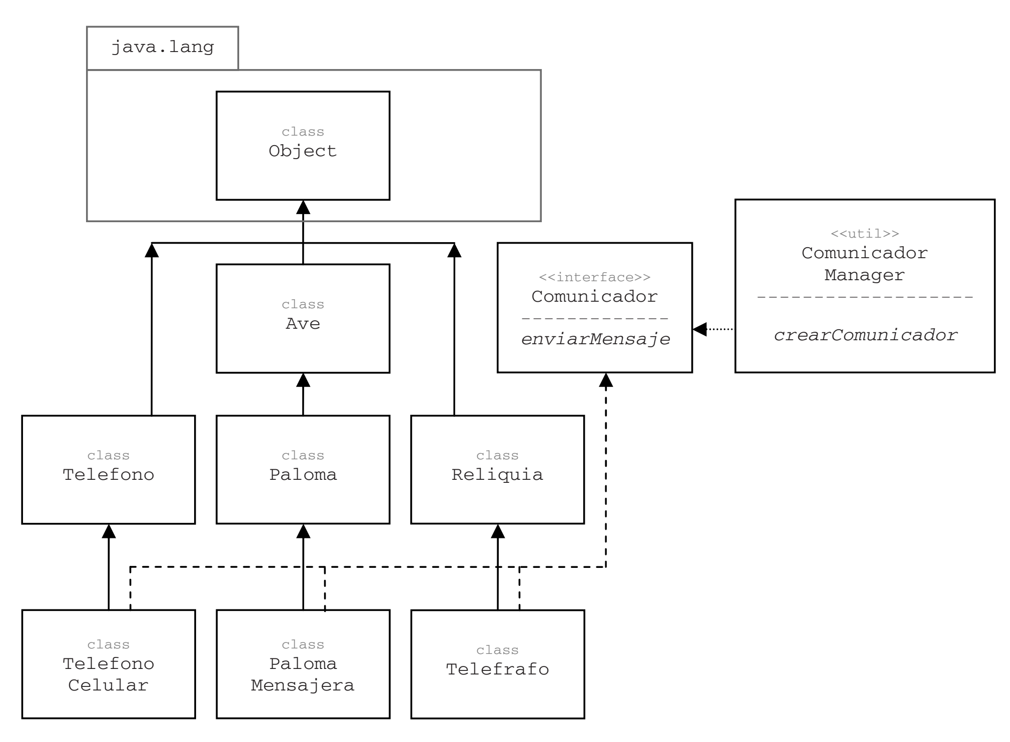


Fig. 14.4 Diagrama de clases de los elementos de comunicación.

Este diagrama debe interpretarse de la siguiente manera: la clase `TelefonoCelular` hereda de la clase `Telefono`. La clase `PalomaMensajera` hereda de la clase `Paloma` y esta, a su vez, hereda de `Ave`. La clase `Telegrafo` hereda de `Reliquia`. Todas estas clases directa o indirectamente heredan de la clase `Object` (que está en el paquete `java.lang`).

Las clases `TelefonoCelular`, `PalomaMensajera` y `Telegrado` implementan la *interface* `Comunicador`, de donde heredan el método `enviarMensaje`. La clase `ComunicadorManager` crea una instancia de `Comunicador` (que en realidad será una instancia de cualquiera de las clases que implementan esta *interface* ya que las *interfaces* no se pueden instanciar).

14.4.2 El patrón de diseño de la factoría de objetos

El método `crearComunicador` de la clase `ComunicadorManager` nos permitió obtener una instancia de `Comunicador` sin tener que *harcodear* un tipo de comunicador en particular. Gracias a este método, nuestro programa (el método `main`) quedó totalmente separado de la implementación concreta del comunicador.

La transición de tecnología que implica pasar de una paloma mensajera a un teléfono celular no ocasionó ningún efecto negativo ya que el programa está totalmente desacoplado de dicha implementación.

Decimos entonces que el método `crearComunicador` permite "fabricar objetos comunicadores". Llamamos a estos métodos "factorías de objetos" o *factory methods*.

14.4.3 Abstracción a través de interfaces

Las *interfaces* ayudan a incrementar el nivel de abstracción tanto como sea necesario porque permiten tener múltiples "vistas" de una misma clase.

Por ejemplo, un objeto de la clase `PalomaMensajera` puede "ser visto" (interprétese "puede asignarse a una variable de tipo...") como un objeto de esta misma clase o bien puede "ser visto" como un objeto de la clase `Paloma` o `Ave` o como un objeto de la clase `Object`. Además, como `PalomaMensajera` implementa la *interface* `Comunicador`, también puede "ser visto" como un objeto de este tipo. Si la clase `PalomaMensajera` hubiera implementado más *interfaces*, entonces sus objetos podrían verse como objetos de cualquiera de estas.

Para estudiar esto analizaremos las siguientes preguntas:

- ¿Podría el lector ordenar un conjunto de valores numéricos enteros? Por supuesto que sí. El conjunto de los números enteros tiene un orden natural; por lo tanto, nadie dudaría en colocar al número 2 antes que el 3, al 3 antes que el 4, etcétera.
- ¿Podría el lector ordenar un conjunto de cadenas de caracteres que representan nombres de personas? Claro que sí. Lo lógico e intuitivo sería ordenarlos alfabéticamente; por lo tanto, el nombre "Alberto" precedería al nombre "Juan" y este precedería al nombre "Pablo".
- ¿Podría el lector ordenar un conjunto de alumnos de una escuela? Este caso no es tan claro como los anteriores. Antes de ordenar a los alumnos deberíamos definir un criterio de precedencia. Por ejemplo, si tomamos como criterio de precedencia "la edad del alumno" entonces primero deberíamos ubicar a los más jóvenes y luego a los mayores. Claro que también podríamos definir como criterio de precedencia "la nota promedio". En este caso, primero ubicaríamos a los que tienen menor nota promedio y luego a los que tienen una mejor calificación.

Ahora, desde el punto de vista del programador que tiene que desarrollar un método para ordenar un conjunto de objetos: ¿Deberíamos preocuparnos por el criterio de precedencia de los elementos del conjunto o sería mejor abstraernos y deslindar en los mismos objetos la responsabilidad de decidir si preceden o no a otros objetos de su misma especie? Desde este punto de vista, nuestra postura debería ser: "yo puedo ordenar cualquier conjunto de objetos siempre y cuando cada elemento del conjunto me pueda decir si precede o no a cualquiera de los otros elementos".

14.4.4 La interface `Comparable`

Java provee la *interface* `Comparable` cuyo código fuente (abreviado) vemos a continuación:

```
public interface Comparable<T>
{
   public int compareTo(T obj);
}
```

Esta *interface* define un único método que recibe un objeto como parámetro y debe retornar un valor entero mayor, menor o igual a cero según resulte la comparación entre los atributos de la instancia (`this`) y los del parámetro `obj`.

Es decir, si vamos a implementar la *interface* `Comparable` en la clase `Alumno` y tomamos como criterio de comparación el atributo `edad` entonces, dados dos alumnos `a` y `b` tal que `a.edad` es menor que `b.edad`, será: `a.compareTo(b)<0`.

La *interface* `Comparable` es genérica en `T` para validar en tiempo de compilación que no se intente comparar elementos de diferentes tipos de datos.

Ahora definiremos la clase `Alumno` con los atributos `nombre`, `edad` y `notaPromedio` e implementaremos la *interface* `Comparable<Alumno>` para determinar el orden de precedencia entre dos alumnos en función de su edad.

```
package libro.cap14.interfaces;

public class Alumno implements Comparable<Alumno>
{
   private String nombre;
   private int edad;
   private double notaPromedio;

   // constructor
   public Alumno(String nom, int e, double np)
   {
      this.nombre = nom;
      this.edad = e;
      this.notaPromedio = np;
   }

   // metodo heredado de la interface Comparable
   public int compareTo(Alumno otroAlumno)
   {
      return this.edad - otroAlumno.edad;
   }

   public String toString()
   {
      return nombre+", "+edad+", "+notaPromedio;
   }

   // :
   // setters y getters
   // :
}
```

El método `compareTo`, debe retornar un valor mayor, menor o igual a cero según resulte la comparación entre la edad de la instancia y la edad del parámetro `otroAlumno`.

Si “nuestra” edad (la variable de instancia `edad`) es mayor que la edad del `otroAlumno` entonces la diferencia entre ambas edades será positiva y estaremos retornando un valor mayor que cero para indicar que “somos mayores” que el `otroAlumno`. Si ambas edades son iguales entonces la diferencia será igual a cero y si la edad de `otroAlumno` es mayor que la “nuestra” entonces retornaremos un valor menor que cero.

Ahora desarrollaremos una clase utilitaria `Util` con un método estático `ordenar`. Este método recibirá un *array* de objetos “comparables”. Con esto, podremos ordenarlos sin problema aplicando (por ejemplo) el algoritmo de “la burbuja” optimizado.

```
package libro.cap14.interfaces;

public class Util
{
   public static void ordenar(Comparable arr[])
   {
      boolean ordenado = false;
      while( !ordenado )
      {
         ordenado = true;
         for(int i = 0; i<arr.length-1; i++)
         {
            if(arr[i+1].compareTo(arr[i])<0)
            {
               Comparable aux = arr[i];
               arr[i] = arr[i+1];
               arr[i+1] = aux;
               ordenado = false;
            }
         }
      }
   }
}
```

En el método `ordenar` recibimos un `Comparable[];` por lo tanto, cada elemento del *array* puede responder sobre si precede o no a otro objeto de su misma especie.

Ejemplo: Ordenar un *array* de alumnos.

Para finalizar podemos hacer un programa donde declaramos un *array* de alumnos, lo ordenamos y lo mostramos ordenado por pantalla.

```
package libro.cap14.interfaces;

public class TestOrdenar
{
   public static void main(String[] args)
   {
      // definimos un array de alumnos
      Alumno arr[] = { new Alumno("Juan",20,8.5)
                     , new Alumno("Pedro",18,5.3)
                     , new Alumno("Alberto",19,4.6) };

      // lo ordenamos
      Util.ordenar(arr);

      // lo mostramos ordenado
      for( int i = 0; i < arr.length; i++ )
      {
         System.out.println(arr[i]);
      }
   }
}
```

La salida de este programa será:

```
Pedro, 18, 5.3
Alberto, 19, 4.6
Juan, 20, 8.5
```

lo que demuestra que el método `ordenar` ordenó a los alumnos contenidos en el *array* en función de su atributo `edad` porque así lo definimos en la clase `Alumno`.

En el siguiente diagrama podemos repasar la relación que existe entre las clases `Alumno`, `Util` y la *interface* `Comparable`.

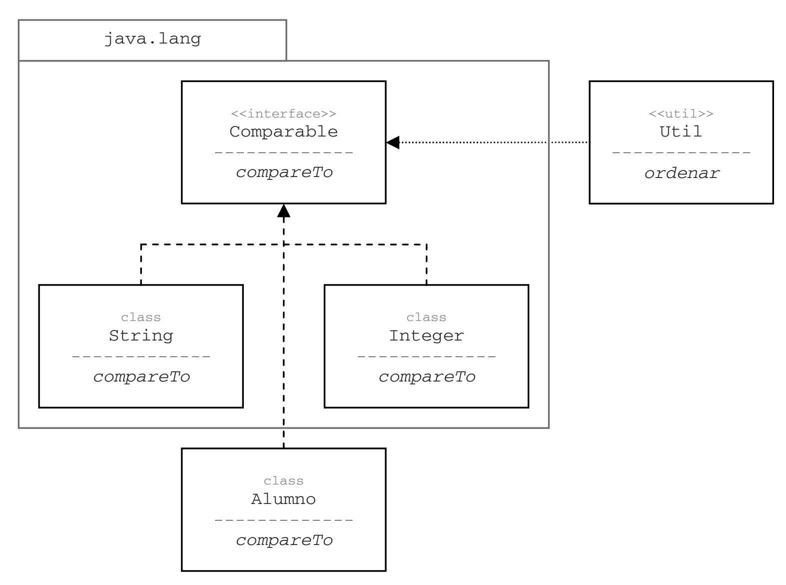


Fig. 14.5 Relación entre las clases `Util`, `Alumno` y la *interface* `Comparable`.

Vemos que la clase `Alumno` implementa la *interface* `Comparable` y sobrescribe el método `compareTo`. Por otro lado, la clase `Util` tiene un método `ordenar` que ordena un *array* de elementos comparables. No importa si estos elementos son alumnos u objetos de otro tipo. En el método `ordenar`, "vemos" a los elementos del *array* como "elementos comparables" (de tipo `Comparable`).

En el diagrama también representamos las clases `String` e `Integer`. Estas clases son provistas por Java en el paquete `java.lang` e implementan la *interface* `Comparable`; por lo tanto, nuestro método `ordenar` también podrá ordenar `String[]` e `Integer[]` ya que ambos tipos de *arrays* contienen elementos comparables.

Ampliaremos el método `main` de la clase `TestOrdenar` para ordenar también un *array* de `String` y otro de `Integer`.

```
package libro.cap14.interfaces;

public class TestOrdenar
{
   public static void main(String[] args)
   {
      // definimos, ordenamos y mostramos un array de alumnos
      Alumno arr[] = { new Alumno("Juan",20,8.5)
                     , new Alumno("Pedro",18,5.3)
                     , new Alumno("Alberto",19,4.6) };
```

```
      Util.ordenar(arr);
      muestraArray(arr);

      // definimos, ordenamos y mostramos un array de strings
      String[] arr2 = { "Pablo","Andres","Marcelo" };
      Util.ordenar(arr2);
      muestraArray(arr2);

      // definimos, ordenamos y mostramos un array de integers
      Integer[] arr3 = { new Integer(5)
                       , new Integer(3)
                       , new Integer(1) };
      Util.ordenar(arr3);
      muestraArray(arr3);
   }

   private static void muestraArray(Comparable arr[])
   {
      for( int i=0; i<arr.length; i++ )
      {
         System.out.println(arr[i]);
      }
   }
}
```

En este ejemplo utilizamos el método `Util.ordenar` para ordenar *arrays* de diferentes tipos: `Alumno`, `String` e `Integer`. Todos estos tipos de datos tienen algo en común: son comparables y, por lo tanto, tienen el método `compareTo` que utilizamos en el método `ordenar` para ordenarlos.

Volviendo a la consigna original, logramos que nuestro método `ordenar` pueda ordenar objetos de cualquier tipo de datos siempre y cuando estos sean comparables.

14.4.5 Desacoplar aún más

En el ejemplo anterior, implementamos la *interface* `Comparable` en la clase `Alumno` y definimos como criterio de precedencia la edad de los alumnos. Tomando en cuenta este criterio, un alumno precede a otro si el valor de su atributo `edad` es menor que el valor del mismo atributo del otro alumno.

Si decidimos cambiar el criterio y considerar que un alumno precede a otro según el orden alfabético de su nombre, tendremos que reprogramar el método `compareTo` en la clase `Alumno`. Esto, además de la necesidad de modificar nuestro código, implica perder el criterio de comparación anterior ya que solo podemos sobrescribir el método `compareTo` una única vez.

Sin embargo, ¿sería descabellado pretender ordenar un conjunto de alumnos según su nombre y también pretender ordenarlos según su edad y hasta pretender ordenarlos según su nota promedio?

Al implementar la *interface* `Comparable` en `Alumno` estamos “hardcodeando” el criterio de precedencia. Quizás una mejor solución sería definir una clase abstracta `Criterio` que defina un método abstracto `comparar`, el cual reciba dos parámetros del mismo tipo y retorne un entero mayor, igual o menor que cero según resulte la comparación entre estos.

El siguiente diagrama nos ayudará a comprender mejor esta idea.

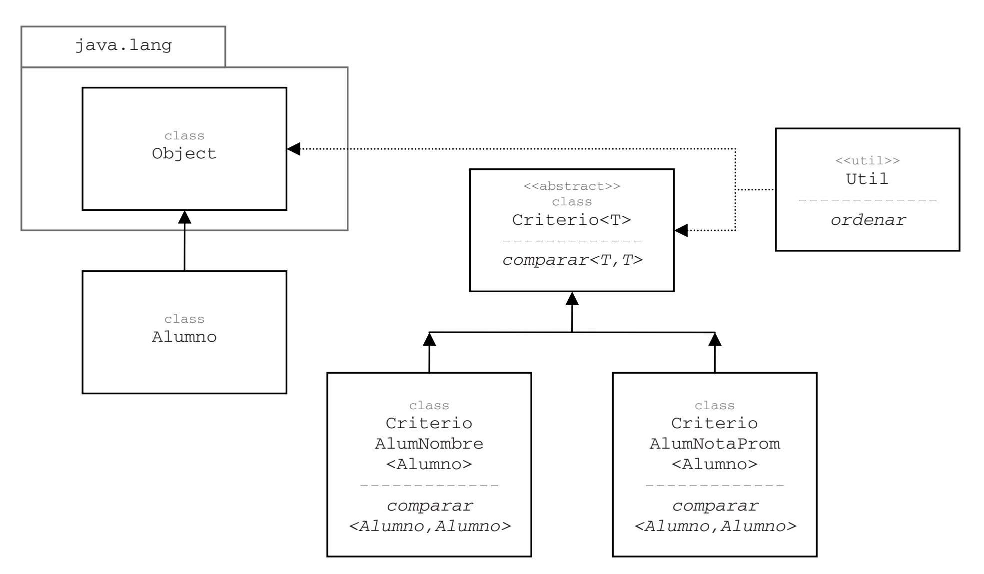


Fig. 14.6 Implementaciones de la clase `Criterio<T>`.

En esta nueva versión, ya no necesitamos imponer que las clases implementen `Comparable` para que nuestro método `Util.ordenar` pueda ordenar sus objetos.

En el diagrama vemos que `Alumno` hereda de `Object` y no implementa la *interface* `Comparable`. La nueva versión del método `ordenar` de la clase `Util` recibe un `Object[]` y un `Criterio`, siendo este parámetro una instancia de alguna implementación concreta ya que `Criterio` es una clase abstracta. Veamos la clase `Criterio`.

```
package libro.cap14.interfaces.criterios;

public abstract class Criterio<T>
{
   public abstract int comparar(T a, T b);
}
```

La clase `Criterio` es genérica en `T`, lo que nos permite asegurar que los dos objetos que vayamos a comparar sean del mismo tipo de datos.

Veamos ahora dos implementaciones de `Criterio<T>`: una para comparar alumnos por su nombre y otra para compararlos por su nota promedio.

```
package libro.cap14.interfaces.criterios;

import libro.cap02.interfaces.Alumno;

// heredamos de Criterio especializado en Alumno
public class CriterioAlumNombre extends Criterio<Alumno>
{
   public int comparar(Alumno a, Alumno b)
   {
      return a.getNombre().compareTo(b.getNombre());
   }
}
```

```
package libro.cap14.interfaces.criterios;

import libro.cap14.interfaces.Alumno;

public class CriterioAlumNotaProm extends Criterio<Alumno>
{
   public int comparar(Alumno a, Alumno b)
   {
      double diff = a.getNotaPromedio()-b.getNotaPromedio();
      return diff>0 ? 1: diff <0 ? -1 : 0;
   }
}
```

En este método no podemos retornar la diferencia entre las dos notas promedio porque estos valores son de tipo `double`. En la línea:

```
return diff>0 ? 1: diff <0 ? -1 : 0;
```

hacemos un doble *if-inline*. Primero preguntamos si `diff` es mayor que cero. Si es así retornamos 1. Si no, preguntamos si `diff` es menor que cero. En este caso retornamos -1 y, si no, retornamos 0.

Veamos ahora la clase `Util`, donde modificamos el método `ordenar` para que reciba un `Object[]` y una implementación de `Criterio`.

```
package libro.cap14.interfaces.criterios;

public class Util
{
   public static void ordenar(Object arr[], Criterio cr)
   {
      boolean ordenado = false;
      while( !ordenado )
      {
         ordenado = true;
         for( int i = 0, j = arr.length - 1; i <= j; i++, j-- )
         {
            // ahora la decision sobre "quien precede a quien"
            // la toma la instancia de Criterio cr
            if( cr.comparar(arr[i+1],arr[i]) < 0 )
            {
               Object aux = arr[i];
               arr[i] = arr[i + 1];
               arr[i + 1] = aux;
               ordenado = false;
            }
         }
      }
   }

   public static void imprimir(Object arr[])
   {
      for(int i=0; i<arr.length; i++)
      {
         System.out.println(arr[i]);
      }
   }
}
```

Aprovechamos e incluimos en esta clase un método estático `imprimir` que permite imprimir todos los elementos de un `Object[]`.

En el método `ordenar`, en la línea:

```
if( cr.comparar(arr[i+1],arr[i]) < 0 )
```

invocamos al método `comparar` del objeto `cr` que, en función de su implementación, podrá determinar si `arr[i+1]` precede o no a `arr[i]`.

Para terminar, veremos un ejemplo donde definimos un `Alumno[]` y lo imprimimos, ordenado primero por `nombre` y luego por `notaPromedio`.

```
package libro.cap14.interfaces.criterios;

import libro.cap14.interfaces.Alumno;

public class TestCriterio
{
   public static void main(String[] args)
   {
      Alumno arr[] = { new Alumno("Martin", 25, 7.2)
                      ,new Alumno("Carlos", 23, 5.1)
                      ,new Alumno("Anastasio", 20, 4.8) };

      // ordenamos el array segun el nombre de los alumnos
      Util.ordenar(arr,new CriterioAlumNombre());
      Util.imprimir(arr);

      // ordenamos el array segun la nota promedio de los alumnos
      Util.ordenar(arr,new CriterioAlumNotaProm());
      Util.imprimir(arr);
   }
}
```

■

14.4.6 La interface `Comparator`

Para que el código sea más estándar, en lugar de utilizar nuestra propia clase abstracta `Criterio` utilizaremos la *interface* que Java provee para el mismo fin: `Comparator`.

Los cambios que debemos aplicar en el ejemplo anterior para utilizar esta *interface* son los siguientes:

1. Eliminar la clase `Criterio`.
2. Hacer que las clases `CriterioAlumNombre` y `CriterioAlumNotaProm` implementen la *interface* `Comparator` y sobrescriban adecuadamente el método `compare`.
3. El método `Util.ordenar` debe recibir un `Object[]` y una instancia de alguna implementación concreta de `Comparator`.

Dejo a cargo del lector, la aplicación de estas modificaciones.

14.5 Colecciones de objetos

Genéricamente, llamamos "colección" a cualquier conjunto de objetos. Un `String[]` es una colección de cadenas, un `Integer[]` es una colección de objetos `Integer` y un `Object[]` es una colección de objetos de cualquier tipo porque, como ya sabemos, todos las clases heredan de la clase base `Object`.

Java provee una *interface* `Collection`; por lo tanto, en general, cuando hablamos de "colección" es porque nos estamos refiriendo a un objeto cuya clase implementa esta *interface*.

Existen varias clases que implementan la *interface* `Collection`. Las más utilizadas son `ArrayList` y `Vector`.

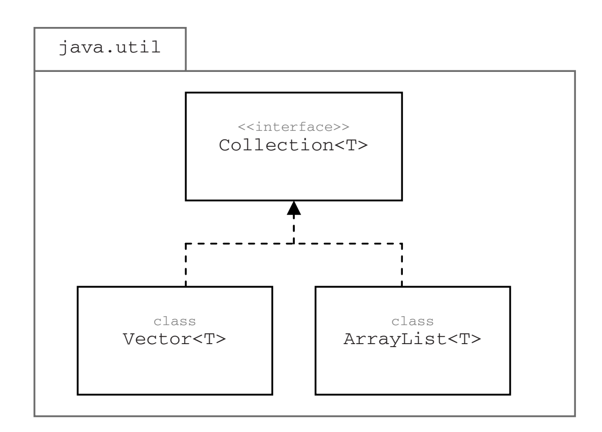


Fig. 14.7 Representación de las clases `Vector` y `ArrayList`.

En el diagrama vemos que tanto la *interface* `Collection` como las dos implementaciones que mencionamos están ubicadas en el paquete `java.util`.

Ejemplo: Uso de la clase `Vector`.

En el siguiente programa, instanciamos un `Vector<String>`, le asignamos algunos valores y luego lo recorremos mostrando su contenido.

```
package libro.cap14.colecciones;

import java.util.Vector;

public class TestVector
{
   public static void main(String[] args)
   {
      // instanciamos un Vector especializado en String
      Vector<String> v = new Vector<String>();

      // le asignamos algunos valores
      v.add("Pablo");
      v.add("Juan");
      v.add("Carlos");
```

```
      String aux;

      // el metodo size indica cuantos elementos contiene el vector
      for(int i=0; i<v.size(); i++ )
      {
         // el metodo get retorna el i-esimo elemento
         aux = v.get(i);
         System.out.println(aux);
      }
   }
}
```

El lector no debe confundir “*array*” con “Vector”. Un *array* es una estructura de datos primitiva. Nosotros, como programadores, no podemos “programar un *array*”, solo podemos utilizarlo. En cambio, `Vector` es una clase que alguien programó y la incluyó en la biblioteca de clases que provee el lenguaje Java.

Es importante notar que el vector mantiene una lista de objetos en memoria. Quizás internamente utilice un *array* o tal vez utilice algún archivo temporal. No lo sabemos y no nos debería interesar saberlo ya que eso es parte de la implementación de la clase. Claro que si utilizamos `Vector` en nuestro programa y luego nos damos cuenta de que esta clase no es tan eficiente como hubiéramos esperado, para cambiarla tendremos que “buscar y reemplazar” la palabra “Vector” por el nombre de la clase que utilizaremos en su lugar.

Además, ¿qué sucedería si en nuestro código invocamos métodos de `Vector` para realizar una determinada tarea y luego, al reemplazar `Vector` por otra clase, vemos que esos métodos no existen en la nueva clase? Estos planteamientos no hacen más que reforzar los conceptos que estudiamos en las secciones anteriores, donde hablamos de desacoplamiento y factorías de objetos.

Ahora reformularemos el ejemplo anterior pero considerando que la lista de nombres la obtendremos a través de una clase utilitaria que llamaremos `UNombres`. Esta clase tendrá un método estático `obtenerLista`.

```
package libro.cap14.colecciones;

import java.util.Collection;
import java.util.Vector;

public class UNombres
{
   public static Collection<String> obtenerLista()
   {
      Vector<String> v = new Vector<String>();
      v.add("Pablo");
      v.add("Juan");
      v.add("Carlos");

      return v;
   }
}
```

Notemos que el tipo de datos del valor de retorno del método es `Collection<String>` aunque lo que realmente estamos retornando es un `Vector<String>`. Esto es correcto porque, como vimos más arriba, la clase `Vector` implementa la *interface* `Collection`. Un `Vector` es una `Collection`.

Ahora, en el programa principal podemos obtener la lista de nombres a través del método `UNombres.obtenerLista` de la siguiente manera:

```
package libro.cap14.colecciones;

import java.util.Collection;

public class TestVector
{
   public static void main(String[] args)
   {
      // el metodo obtenerLista retorna una Collection
      Collection<String> coll = UNombres.obtenerLista();

      // iteramos la coleccion de nombres y mostramos cada elemento
      for(String nom: coll)
      {
         System.out.println(nom);
      }
   }
}
```

En esta nueva versión del programa, obtenemos la lista de nombres invocando al método estático `UNombres.obtenerLista`. Este método retorna un `Collection<String>`; por lo tanto, tenemos que asignar su retorno a una variable de este tipo de datos. Claramente, el método `obtenerLista` es un *factory method*.

Luego, para iterar la colección, es decir, recorrer uno a uno sus elementos y mostrarlos, utilizamos un `for` *each*. Este `for` realiza esta tarea y en cada iteración asigna el *i-ésimo* elemento a la variable `nom`.

14.5.1 Cambio de implementación

Supongamos ahora que, efectivamente, la clase `Vector` no nos termina de convencer y decidimos reemplazarla por `ArrayList`. Este cambio lo aplicaremos en el método `obtenerLista` de la clase `UNombres`.

```
package libro.cap14.colecciones;

import java.util.ArrayList;
import java.util.Collection;

public class UNombres
{
   public static Collection<String> obtenerLista()
   {
      // Vector<String> v = new Vector<String>();
      ArrayList<String> v = new ArrayList<String>();
      v.add("Pablo");
      v.add("Juan");
      v.add("Carlos");

      return v;
   }
}
```

Como `ArrayList` implementa `Collection`, el reemplazo de una clase por otra no generará ningún impacto en el programa principal. En el `main` no nos interesa saber si `UNombres.obtenerLista` retorna un `Vector` o un `ArrayList` ya que interactuamos con la colección de objetos a través de su *interface* y no de su implementación.

14.5.2 El método `Collections.sort`

Java provee el método estático `sort` dentro de la clase utilitaria `Collections`. Este método permite ordenar una colección de objetos en función de un criterio que estará dado por una implementación de la *interface* `Comparator` estudiada más arriba.

La colección de objetos que se va a ordenar debe estar contenida en una lista. Esto es: una implementación de la *interface* `List`, que es una *subinterface* de `Collection`. Las clases `Vector` y `ArrayList` (por ejemplo) implementan esta *interface*.

Veamos un diagrama de clases que nos permita resumir estas ideas:

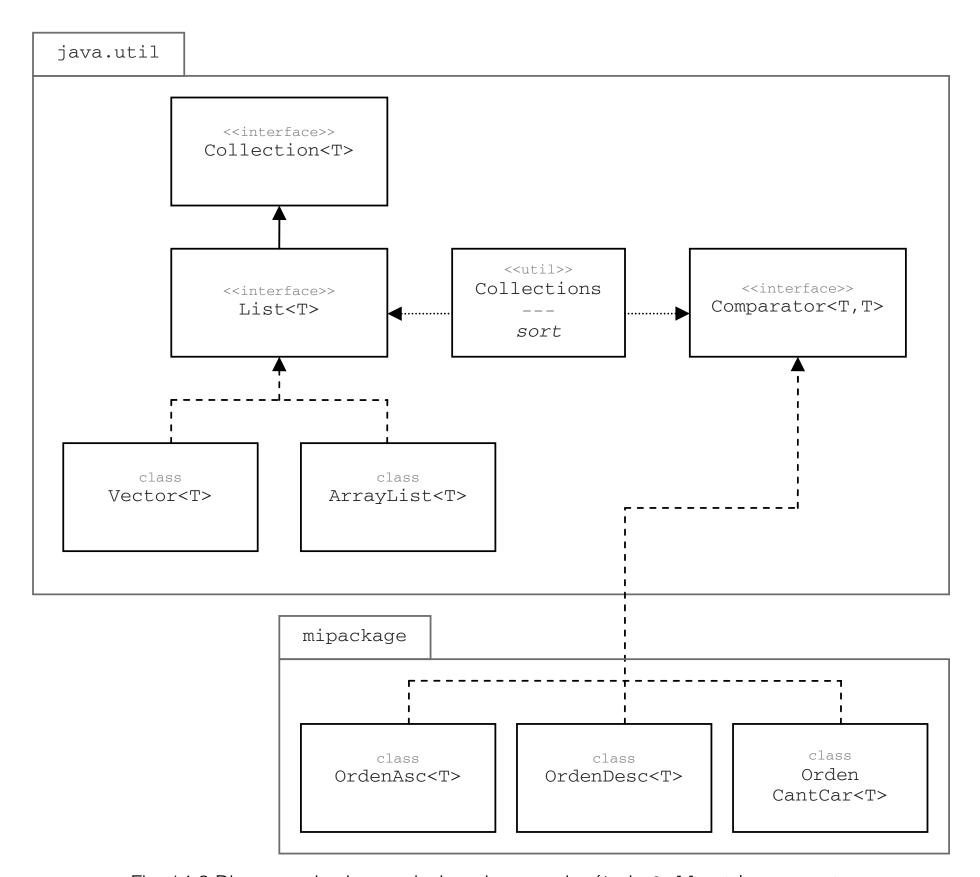


Fig. 14.8 Diagrama de clases relacionadas con el método `Collections.sort`.

En el diagrama vemos que la *interface* `List` extiende a la *interface* `Collection` y que las clases `Vector` y `ArrayList` la implementan. También vemos, en otro paquete, las clases `OrdenAsc`, `OrdenDesc` y `OrdenCantCar`, todas implementaciones de `Comparator`. Finalmente, la clase `Collections` con su método `sort` utiliza una implementación de `List` y una implementación de `Comparator` para realizar su tarea.

Veamos un programa que, utilizando `Collections.sort`, ordene un `ArrayList<String>` de diferentes maneras.

```
package libro.cap14.sort;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TestSort
{
   public static void main(String[] args)
   {
      ArrayList<String> arr = new ArrayList<String>();
      arr.add("Pablo");
      arr.add("Nora");
      arr.add("Rolando");
      arr.add("Analia");
      arr.add("Aldo");
      arr.add("Octaviano");
      arr.add("Luz");

      // orden alfabetico ascendente
      Collections.sort(arr, new OrdenAsc());
      mostrar(arr);
      System.out.println("---");

      // orden alfabetico descendente
      Collections.sort(arr, new OrdenDesc());
      mostrar(arr);
      System.out.println("---");

      // orden por [cantidad de caracteres]+[orden alfabetico]
      Collections.sort(arr, new OrdenCantCar());
      mostrar(arr);
      System.out.println("---");
   }

   public static void mostrar(List<String> lst)
   {
      for(String s:lst)
      {
         System.out.println(s);
      }
   }
}
```

Obviamente, `Collections.sort` ordenará la lista de nombres que contiene `arr` en función de la implementación de `Comparator` que le pasemos como argumento.

Primero veamos la salida del programa:

```
Aldo
Analia
Luz
Nora
Octaviano
```

```
Pablo
Rolando
---
Rolando
Pablo
Octaviano
Nora
Luz
Analia
Aldo
---
Luz
Aldo
Nora
Pablo
Analia
Rolando
Octaviano
---
```

Veamos ahora el código de las clases `OrdenAsc`, `OrdenDesc` y `OrdenCantCar`, todas implementaciones de `Comparator`, que definen el criterio de comparación entre dos cadenas:

Clase	Criterio
`OrdenAsc`	Orden alfabético ascendente
`OrdenDesc`	Orden alfabético descendente
`OrdenCantCar`	Orden por cantidad de caracteres y, a igual cantidad, la precedencia se determinará siguiendo el orden alfabético.

```
package libro.cap14.sort;

import java.util.Comparator;

public class OrdenAsc implements Comparator<String>
{
   public int compare(String s1, String s2)
   {
      return s1.compareTo(s2);
   }
}
```

■

```
package libro.cap14.sort;

import java.util.Comparator;

public class OrdenDesc implements Comparator<String>
{
   public int compare(String s1, String s2)
   {
      return -s1.compareTo(s2);
   }
}
```

■

```
package libro.cap14.sort;

import java.util.Comparator;

public class OrdenCantCar implements Comparator<String>
{
   public int compare(String s1, String s2)
   {
      int x = s1.length()-s2.length();
      return x!=0?x:s1.compareTo(s2);
   }
}
```

`Collections.sort` ordena los elementos de la lista utilizando el algoritmo de ordenamiento *quick sort*. Este algoritmo es altamente eficiente y lo estudiaremos en detalle en el Capítulo 16.

14.6 Excepciones

Las excepciones constituyen un mecanismo de tratamiento de error a través del cual los métodos pueden finalizar abruptamente ante la ocurrencia de una situación anómala que imposibilite su normal desarrollo.

El siguiente ejemplo ilustra una situación típica en la que debemos utilizar excepciones: supongamos que tenemos una clase `Aplicacion` y esta tiene un método `login` que recibe como parámetro dos cadenas: `usrname` y `password`. El método retorna una instancia de `Usuario`, siendo esta una clase con los atributos del usuario que está intentando *loguearse* (nombre, dirección, email, etc.), o `null` si `usrname` y/o `password` son incorrectos.

En el siguiente código, intentamos *loguear* un usuario `"juan"` con un *password* `"juan123sito"`.

```
// instanciamos la clase Aplicacion
Aplicacion app = new Aplicacion();

// intentamos el login
Usuario u = app.login("juan", "juan123sito");

// si los datos no son correctos...
if( u==null )
{
    System.out.println("usuario y/o password incorrectos");
}
else
{
    System.out.println("Felicidades, login exitoso.");
    System.out.println("Nombre: "+u.getNombre());
    System.out.println("Email: "+u.getEmail());
}
```

Este código es muy claro y no necesita ninguna explicación adicional. Sin embargo, el

método `login`, así como está planteado, tiene un importante error de diseño: el método retorna una instancia de `Usuario` si el *login* fue exitoso o `null` si el nombre de usuario y/o el *password* provistos como argumentos no son correctos, pero no se contempla la posibilidad de que, por algún factor externo, el método pueda fallar.

Supongamos que para verificar el *login* el método tiene que acceder a una base de datos y resulta que en ese momento la base de datos está caída. ¿Qué valor debería retornar el método `login`? Si retorna `null` entonces quien lo llamó interpretará que el *username* o el *password* que ingresó son incorrectos. En este caso, el método no debe retornar nada, simplemente debe finalizar arrojando una excepción.

Para probar el ejemplo anterior, en lugar de utilizar una base de datos (porque el tema excede el alcance de este libro), utilizaremos un archivo de propiedades en el cual tendremos definidos los valores de las propiedades del usuario: `usrname`, `password`, `nombre` y `email`.



Un archivo de propiedades es un archivo de texto donde cada línea define una propiedad con su correspondiente valor.

Un archivo de propiedades es un archivo de texto donde cada línea define una propiedad con su correspondiente valor. En nuestro caso llamaremos al archivo `usuario.properties`, el cual debe estar ubicado en la raíz del proyecto de Eclipse y su contenido será el siguiente:

```
usrname=juan
password=juan123sito
nombre=Juan Cordero de Dios
email=juan@juancho.com
```

Teniendo el archivo de propiedades creado y correctamente ubicado, podemos codificar la clase `Aplicacion` y el método `login`.

```
package libro.cap14.excepciones;

import java.io.FileInputStream;
import java.util.Properties;

public class Aplicacion
{
   public Usuario login(String usrname, String password)
   {
      try
      {
         // abrimos el archivo de propiedades para lectura
         FileInputStream fis = new FileInputStream("usuario.properties");

         // cargamos el archivo de propiedades en un objeto tipo Properties
         Properties p = new Properties();
         p.load(fis);

         // leemos el valor de la propiedad usrname
         String usr = p.getProperty("usrname");

         // leemos el valor de la propiedad password
         String pwd = p.getProperty("password");

         // definimos la variable de retorno
         Usuario u = null;
```

```
            // si coinciden los datos proporcionados con los leidos
            if( usr.equals(usrname) && pwd.equals(password) )
            {
               // instanciamos y "seteamos" todos los datos
               u = new Usuario();
               u.setUsrname(usr);
               u.setPassword(pwd);
               u.setNombre(p.getProperty("nombre"));
               u.setEmail(p.getProperty("email"));
            }

            // retornamos la instancia o null si no entramos al if
            return u;
         }
         catch(Exception ex)
         {
            // cualquier error "salgo por excepcion"
            throw new RuntimeException("Error verificando datos", ex);
         }
      }
   }
```

En este ejemplo vemos que todo el código del método `login` está encerrado dentro de un gran bloque `try`. Decimos entonces que “intentamos ejecutar todas esas líneas” y suponemos que todo saldrá bien. Incluso el `return` del método está ubicado como última línea del `try`. Si algo llegase a fallar, entonces la ejecución del código saltará automáticamente a la primera línea del bloque `catch`. Dentro del `catch` “arrojamos una excepción” indicando un breve mensaje descriptivo y adjuntando la excepción original del problema.

Cuando trabajamos con excepciones tratamos el código como si no fuese a ocurrir ningún error. Esto nos permite visualizar un código totalmente lineal y mucho más claro, y ante la ocurrencia del primer error (excepción) “saltamos” al bloque `catch` para darle un tratamiento adecuado o bien (como en nuestro ejemplo) para arrojar una nueva excepción que deberá tratar quien haya invocado a nuestro método.

Veamos ahora el programa que utiliza el método `login` de la clase `Aplicacion`.

```
package libro.cap14.excepciones;

public class TestLogin
{
   public static void main(String[] args)
   {
      try
      {
         Aplicacion app = new Aplicacion();

         // intentamos el login
         Usuario u = app.login("juan","juan123sito");

         // mostramos el resultado
         System.out.println(u);
```

```
      }
      catch(Exception ex)
      {
         // ocurrio un error
         System.out.print("Servicio temporalmente interrumpido: ");
         System.out.println(ex.getMessage());
      }
   }
}
```

En el programa intentamos el *login* como si todo fuera a funcionar perfecto y ante cualquier error "saltamos" al `catch` para mostrar un mensaje de error.

Le recomiendo al lector realizar las siguientes pruebas:

1. Ejecutar el programa así como está y verificar el resultado. En este caso aparecerán en consola todos los datos del usuario.
2. Cambiar el *password* y/o el *username* en el archivo, ejecutar el programa y verificar el resultado: en este caso aparecerá en consola el mensaje: `null`.
3. Mover a otra carpeta el archivo `usuario.properties` y volver a ejecutar el programa. En este caso aparecerá un mensaje de error que indica que el servicio se encuentra temporalmente interrumpido y que hubo un error al verificar los datos. Esto no quiere decir que los datos proporcionados sean incorrectos; simplemente no se los pudo verificar.

14.6.1 Errores lógicos vs. errores físicos

Podemos diferenciar entre estos dos tipos de errores:

1. Errores físicos.
2. Errores lógicos (que en realidad no son errores).

En nuestro caso, errores físicos podrían ser:

- Que no se pueda abrir el archivo de propiedades.
- Que dentro del archivo de propiedades no se encuentre definida alguna de las propiedades cuyo valor estamos intentando leer.

Un error lógico sería:

- Que el *username* y/o el *password* proporcionados como argumentos sean incorrectos, aunque esto en realidad no es un error: la situación ya está contemplada dentro de los escenarios posibles de la aplicación.

14.6.2 Excepciones declarativas y no declarativas

Las excepciones pueden ser declarativas o no declarativas. Si un método declara que, llegado el caso, arrojará excepciones, entonces quien lo invoque estará obligado a encerrar la llamada al método dentro de un bloque *try-catch*.

Si un método no declara que arrojará excepciones entonces solo podrá arrojar excepciones no declarativas. Este es un tipo especial de excepción que no obliga al llamador del método a encerrar la llamada dentro de un *try-catch*.

En el ejemplo anterior, trabajamos con excepciones no declarativas ya que en el prototipo del método `login` no especificamos que este podría arrojar una excepción.

`RuntimeException` es una de las excepciones no declarativas provistas con Java.

Podemos arrojar esta excepción sin tener que declararla en el prototipo del método. Así, quien lo llame no estará obligado a usar *try*-catch en la llamada.

Sin embargo, podemos declarar una lista de excepciones que el método podría arrojar. Por ejemplo:

```
public Usuario login(String usrname
                   , String password) throws ErrorFisicoException
```

El prototipo del método indica que, llegado el caso, se arrojará una excepción de tipo `ErrorFisicoException`. Esto obligará al llamador del método `login` a encerrar la llamada dentro de un bloque *try-catch*.

Las excepciones en realidad son instancias de subclases de `Exception`, por lo que podemos programar nuestra excepción `ErrorFisicoException` de la siguiente manera:

```
package libro.cap14.excepciones;

public class ErrorFisicoException extends Exception
{
   public ErrorFisicoException(Exception ex)
   {
      super("Ocurrio un error fisico", ex);
   }
}
```

Veamos la versión modificada del método `login` de la clase `Aplicacion` donde declaramos que podemos arrojar una excepción de tipo `ErrorFisicoException`.

```
package libro.cap14.excepciones;

import java.util.ResourceBundle;

public class Aplicacion
{
   public Usuario login(String usrname
                      , String password) throws ErrorFisicoException
   {
      try
      {
         // :
         // aqui nada cambio... todo sigue igual
         // :
      }
      catch(Exception ex)
      {
         throw new ErrorFisicoException(ex);
      }
   }
}
```

En el `main` ahora estamos obligados a encerrar la llamada al método `login` dentro de un bloque *try-catch*. Si no lo hacemos, entonces no podremos compilar.

```
package libro.cap14.excepciones;

public class TestLogin
{
   public static void main(String[] args)
   {
      try
      {
         Aplicacion app = new Aplicacion();
         Usuario u = app.login("juan","juan123sito");

         System.out.println(u);
      }
      catch(ErrorFisicoException ex)
      {
         // ocurrio un error
         System.out.print("Servicio temporalmente interrumpido: ");
         System.out.println( ex.getMessage() );
      }
   }
}
```

14.6.3 El bloque `try-catch-finally`

El bloque *try-catch* se completa con la sección `finally`, aunque no es obligatoria. Podemos utilizar las siguientes combinaciones:

- *try-catch*
- *try-finally*
- *try-catch-finally*

Cuando utilizamos `finally` Java nos asegura que siempre, suceda lo que suceda, el programa pasará por allí. Veamos algunos ejemplos.

En el siguiente programa, imprimimos la cadena "`Hola, chau !`" dentro del `try` y luego finalizamos el método `main` con la sentencia `return`. Antes de finalizar, el programa ejecutará el código ubicado en la sección `finally`.

```
package libro.cap14.excepciones;

public class Demo1
{
   public static void main(String[] args)
   {
      try
      {
         System.out.println("Hola, chau !");
         return;
      }
      catch(Exception ex)
      {
         System.out.println("Entre al catch...");
```

```
      }
      finally
      {
         System.out.println("Esto sale siempre !");
      }
   }
}
```

La salida será:

```
Hola, chau !
Esto sale siempre !
```

En el siguiente programa, prevemos la posibilidad de "pasarnos de largo" en un *array*, por lo que capturamos una (posible) excepción del tipo `ArrayIndexOutOfBoundException`. Sin embargo, la excepción que se originará será de tipo `NumberFormatException` porque dentro del `try` intentamos convertir a `int` una cadena que no tiene formato numérico.

En este caso, el programa "saldrá por el *throws*" y no entrará al `catch`, pero primero pasará por el `finally`.

```
package libro.cap14.excepciones;

public class Demo2
{
   public static void main(String[] args) throws Exception
   {
      try
      {
         int i = Integer.parseInt("no es una cadena numerica...");
      }
      catch(ArrayIndexOutOfBoundsException ex)
      {
         System.out.println("Entre al catch...");
      }
      finally
      {
         System.out.println("Esto sale siempre !");
      }
   }
}
```

La sección `finally` es el lugar ideal para devolver los recursos físicos que tomamos desde nuestro programa: cerrar archivos, cerrar conexiones con bases de datos, etcétera.

Por último, volvamos al código del método `login` en la clase `Aplicacion`. Dentro de este método utilizamos un `FileInputStream` para abrir el archivo `usuario.properties`. Tal vez el lector haya notado que luego de utilizar el archivo no lo cerramos. En ese momento se omitió ese detalle porque no habíamos explicado la sección `finally`. Sin embargo, ahora que la conocemos podemos mejorar el código del método `login` de la siguiente manera.

```java
package libro.cap14.excepciones;

import java.io.FileInputStream;
import java.util.Properties;

public class Aplicacion
{
   public Usuario login(String usrname, String password)
   {
      FileInputStream fis = null;

      try
      {
         // abrimos el archivo de propiedades para lectura
         fis = new FileInputStream("usuario.properties");

         // cargamos el archivo de propiedades en un objeto tipo Properties
         Properties p = new Properties();
         p.load(fis);

         // leemos el valor de la propiedad usrname
         String usr = p.getProperty("usrname");

         // leemos el valor de la propiedad password
         String pwd = p.getProperty("password");

         // definimos la variable de retorno
         Usuario u = null;

         // si coinciden los datos proporcionados con los leidos
         if( usr.equals(usrname) && pwd.equals(password) )
         {
            // instanciamos y "seteamos" todos los datos
            u = new Usuario();
            u.setUsrname(usr);
            u.setPassword(pwd);
            u.setNombre(p.getProperty("nombre"));
            u.setEmail(p.getProperty("email"));
         }

         // retornamos la instancia o null si no entramos al if
         return u;
      }
      catch(Exception ex)
      {
         ex.printStackTrace();
         throw new RuntimeException("Error verificando datos", ex);
      }
      finally
      {
         try
         {
            if( fis!=null ) fis.close();
         }
         catch(Exception ex)
         {
            ex.printStackTrace();
            throw new RuntimeException(ex);
         }
      }
   }
}
```

En esta versión de `login`, declaramos la variable `fis` afuera del `try` porque necesitamos que sea visible desde el `try` (para instanciarla) y desde el `finally` (para cerrarla).

Como el método `close` de la clase `FileInputStream` arroja una excepción declarativa de tipo `IOException`, estamos obligados a encerrarlo en un *try-catch* dentro del `finally`.

También, antes de lanzar cada `RuntimeException`, invocamos sobre la excepción el método `printStackTrace`. Esto lo analizaremos a continuación.

14.6.4 El método `printStackTrace`

Las excepciones heredan de la clase base `Exception` y, a su vez, esta hereda de la clase `Throwable` el método `printStackTrace`.

Una vez originado el error, `printStackTrace` imprime en la *standard error* (que por defecto es la consola) el estado actual de la pila de llamadas.

El método `printStackTrace` es fundamental para encontrar errores de lógica, por lo que siempre deberíamos incluirlo dentro del `catch` como vemos a continuación.

```
try
{
   // ...
}
catch(Exception ex)
{
   ex.printStackTrace();
   // ...
}
```

14.7 Resumen

En este capítulo estudiamos en detalle todos los conceptos de la programación orientada a objetos: polimorfismo, *interfaces*, encapsulamiento, clases abstractas y demás. Esto nos permitirá, en adelante, manejar un lenguaje más preciso y utilizar estos recursos cuando sea necesario.

En el próximo capítulo, estudiaremos las clases que provee Java para trabajar con estructuras de datos dinámicas lineales.

14.8 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

14.8.1 Mapa conceptual

14.8.2 Autoevaluaciones

14.8.3 Videotutorial

14.8.3.1 Uso del javadoc

14.8.4 Presentaciones*

15

Estructuras de datos dinámicas lineales en Java

Contenido

Objetivos del capítulo

- Conocer las clases con las que Java encapsula las diferentes estructuras de datos.
- Estudiar las *interfaces* `List`, `Map` y sus implementaciones.
- Determinar cuando utilizar una u otra implementación de una misma *interface*.
- Tejer estructuras de datos combinadas.

Competencias específicas

- Aplicar los conocimientos sobre estructuras lineales en la solución de problemas del mundo real con Java.

15.1 Introducción

Como comentamos en los capítulos anteriores, Java se caracteriza por tener una extensa biblioteca de clases que proveen funcionalidad para la más amplia y diversa gama de utilidades.

El objetivo de este capítulo es estudiar las clases provistas por Java que aportan funcionalidad para el manejo de estructuras de datos lineales dinámicas.

15.2 Listas (implementaciones de `List`)

De la misma manera que cuando hablamos de "colecciones" nos referimos a clases que implementen la *interface* `Collection`, cuando hablamos de "listas" nos referimos a implementaciones de la *interface* `List` que se encuentra en el paquete `java.util`.

`List` es una sub*interface* de `Collection` que básicamente agrega el método `listIterator` para retornar una instancia (implementación) de la *interface* `ListIterator`.

Las listas proveen un acceso secuencial a sus elementos ya que la *interface* `ListIterator` define, entre otros, los siguientes métodos.

Método	**Descripción**
`hasNext`	Posicionados en un elemento, indica si existe un próximo elemento.
`hasPrevious`	Posicionados en un elemento, indica si existe un elemento anterior.
`next`	Avanza al siguiente elemento.
`previous`	Retrocede al elemento anterior

Fig. 15.1 Algunos métodos definidos en la *interface* `ListIterator`.

A continuación, vamos a analizar y comparar dos implementaciones de `List`: las clases `ArrayList` y `LinkedList`.

15.2 La clase `ArrayList`

Esta clase permite mantener en memoria una colección dinámica de objetos. Es decir, encapsula una estructura de datos lineal en la que podemos almacenar una cantidad ilimitada de elementos siempre y cuando haya suficiente memoria física disponible.

```
public class DemoArrayList
{
   public static void main(String[] args)
   {
      ArrayList<String> a = new ArrayList<String>();
      a.add("uno");
      a.add("dos");
      a.add("tres");

      for(String x:a)
      {
         System.out.println(x);
      }
   }
}
```

La salida de este programa será:

```
uno
dos
tres
```

En este ejemplo instanciamos un `ArrayList` de `String` y le agregamos tres cadenas. Luego lo iteramos con un `for` *each* para mostrar cada uno de sus elementos.

También podemos iterar el *arraylist* con un `for` tradicional. Para esto, la clase provee los métodos `size` y `elementAt` que retornan, respectivamente, la cantidad de elementos que contiene la colección y su *i-ésimo* elemento.

```
      // :
      for( int i=0; i<a.size(); i++ )
      {
         String x = a.elementAt(i);
         System.out.println(x);
      }
   }
}
```

El método `add` está sobrecargado. Si le pasamos únicamente el elemento que queremos agregar entonces lo agregará al final, pero también podemos especificar la posición en la que queremos que se inserte el elemento. Así, podemos modificar el programa anterior de la siguiente manera:

```
public class DemoArrayList2
{
   public static void main(String[] args)
   {
      ArrayList<String> a = new ArrayList<String>();
      a.add(0,"uno");
      a.add(0,"dos");
      a.add(0,"tres");

      for(String x:a)
      {
         System.out.println(x);
      }
   }
}
```

Aquí, cada elemento que agregamos desplazará una posición “hacia atrás” a los demás. Por lo tanto, la salida será:

```
tres
dos
uno
```

La clase define una gran cantidad de métodos adicionales. Algunos de ellos son los siguientes.

Método	**Descripción**
`clear`	Elimina todos los elementos de la colección.
`addAll`	Agrega todos los elementos de una colección que recibe como parámetro.
`contains`	Retorna `true` o `false` según la colección contenga o no al objeto que recibe como parámetro.
`remove`	Elimina el objeto ubicado en la *i-ésima* posición de la colección.
`set`	Reemplaza el objeto ubicado en la *i-ésima* posición de la colección por otro que recibe como parámetro.

Fig. 15.2 Algunos métodos definidos en la clase `ArrayList`.

15.2.2 La clase `LinkedList`

`LinkedList` es la implementación de una lista doblemente enlazada. Veamos un ejemplo.

```
public class DemoLinkedList
{
   public static void main(String[] args)
   {
      LinkedList<String> lst = new LinkedList<String>();
      lst.add("John");
      lst.add("Paul");
      lst.add("George");
      lst.add("Ringo");

      ListIterator<String> aux = lst.listIterator();

      // recorremos desde el primero hasta el ultimo
      while( aux.hasNext() )
      {
         String b = aux.next();
         System.out.println(b);
      }

      System.out.println("---");

      // recorremos desde el ultimo hasta el primero
      while( aux.hasPrevious() )
      {
         String b = aux.previous();
         System.out.println(b);
      }
   }
}
```

La salida de este programa será:

```
John
Paul
George
Ringo
---
Ringo
George
Paul
John
```

15.2.3 Comparación entre `ArrayList` y `LinkedList`

Si comparamos las API de estas dos clases veremos que coinciden en una gran cantidad de métodos y esto se debe a que ambas son implementaciones de las *interfaces* `Collection` y `List`.

Tal vez el lector se esté preguntando: si ambas clases proveen los mismos métodos, ¿entonces qué sentido tiene que existan las dos?

La respuesta a esta pregunta es la siguiente: funcionalmente, las dos clases permiten mantener una colección de objetos en memoria, accederlos directa o secuencialmente, remover uno, algunos o todos sus elementos, etcétera. Sin embargo, cada una está implementada con una estructura de datos diferente, lo que hace que, según sea el caso, utilizar una u otra resulte ser más o menos eficiente.

La implementación de `ArrayList` está basada en el uso de un *array* que se irá redimensionando en la medida en que todas sus posiciones vayan siendo utilizadas; algo parecido a la clase `MiColeccion` que analizamos en el capítulo anterior. En cambio, la implementación de `LinkedList` está basada en una estructura de lista doblemente enlazada como las que estudiamos en el capítulo de estructuras de datos dinámicas lineales.

La diferencia entre ambas clases es fundamental ya que, dependiendo del uso que necesitemos darle, utilizar una u otra hará que nuestro programa tenga mejor o peor rendimiento.

A continuación, desarrollaremos una clase que nos permitirá demostrar empíricamente la afirmación anterior.

15.2.4 Desarrollo de la clase `Performance`

Desarrollaremos la clase `Performance` para medir el rendimiento de determinados métodos, algoritmos y/o procesos. La utilizaremos de la siguiente manera.

```
Performance p = new Performance();
unProcesoQueQuieroMedir();
System.out.println(p);
```

La salida será algo así:

```
15410 milisegundos (0 minutos, 15 segundos)
```

Cada instancia de `Performance` permitirá medir el tiempo transcurrido entre la llamada a su constructor y la llamada a su método `toString`.

La estrategia, entonces, será la siguiente: en el constructor tomamos la hora del sistema (la llamaremos `ti`) para asignarla en una variable de instancia. Luego, en el `toString` volveremos a tomar la hora del sistema (la llamaremos `tf`) y la diferencia entre `tf` y `ti` será el tiempo transcurrido entre las invocaciones al constructor y al método `toString` de la instancia de `Performance`.

```
package libro.performance;

public class Performance
{
   private long ti;
   private long tf;
   private boolean stoped=false;

   public Performance()
   {
      ti = System.currentTimeMillis());
   }

   public String toString()
   {
      if( !stoped )
      {
         tf = System.currentTimeMillis();
         stoped=true;
      }

      long diff = tf - ti;
      long min = diff / 1000 / 60;
      long sec = (diff / 1000) % 60;
      return diff + " milisegundos (" + min + " minutos, " + sec + " segundos)";
   }

   // :
   // setters y getters
   // :
}
```



El método `currentTimeInMillis` de la clase `System` retorna un valor de tipo `long` que indica la cantidad de milisegundos transcurridos entre el 1 de enero de 1970 y el momento en que se lo invocó.

El método `System.currentTimeInMillis` retorna un valor de tipo `long` que indica la cantidad de milisegundos transcurridos entre el 1 de enero de 1970 y el momento en que se lo invocó.

La variable `stoped` es un *flag* que utilizamos para garantizar que el tiempo final `tf` se tomará una única vez. Esto nos permitirá invocar al método `toString` tantas veces como sea necesario sin arruinar la medición.

15.2.5 Introducción al análisis de complejidad algorítmica

Si bien este tema lo trataremos más adelante, considero conveniente exponer algunos conceptos que nos permitan comprender la importancia de hacer una correcta elección al momento de optar entre una u otra estructura de datos.

Dejando de lado las operaciones de entrada y salida, un programa demandará mayor o menor tiempo de ejecución según la cantidad de instrucciones que deba realizar.

Llamemos *t* al tiempo que le lleva a una computadora ejecutar una instrucción. Entonces un programa que ejecuta *n* instrucciones demandará un tiempo estimado *t*n*.

Obviamente, dependiendo de los datos de entrada, el programa entrará (o no) en sentencias condicionales, iterativas, etcétera, por lo que es habitual hablar de “el mejor de los casos” y de “el peor de los casos”.

Por ejemplo, para eliminar un elemento en un *array*, el mejor de los casos sería eliminar el último y el peor de los casos sería eliminar el primero ya que esto implica tener que desplazar “hacia adelante” a todos los demás elementos.

Hecha esta introducción, pasaremos a analizar situaciones en las que una correcta o incorrecta elección entre `ArrayList` y `LinkedList` impactará directamente en el rendimiento de nuestro programa.

15.2.5.1 Acceso aleatorio a los elementos de la colección

Dado que `ArrayList` internamente utiliza un *array*, resulta muy eficiente para proveer acceso directo a cualquiera de sus elementos. Tanto es así que demanda una única instrucción para acceder a cualquiera de los elementos de la colección.

Por el contrario, `LinkedList` utiliza una lista doblemente enlazada. Esto significa que para acceder al *i-ésimo* elemento (nodo) deberá recorrer, uno a uno, los *i*-1 nodos anteriores, lo que le demandará *i*-1 instrucciones.

Evidentemente, en este caso utilizar `ArrayList` resultará más eficiente que utilizar `LinkedList`.

Con el siguiente programa, probaremos la diferencia de rendimiento que, para este caso en particular, existe entre ambas clases.

```
package libro.performance;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ArrayListVsLinkedList
{
   public static void main(String[] args)
   {
      // generamos dos listas de n elementos cada una
      int n = 100000;
      List arrayList = generarLista(ArrayList.class, n);
      List linkedList = generarLista(LinkedList.class, n);

      // recorremos la lista y accedemos al i-esimo elemento
      Performance p1 = new Performance();
      for( int i=0; i<n; i++){ arrayList.get(i); }
      System.out.println(p1);

      // recorremos la lista y accedemos al i-esimo elemento
      Performance p2 = new Performance();
      for( int i=0; i<n; i++){ linkedList.get(i); }
      System.out.println(p2);
   }
```

```
   private static List generarLista(Class impleList, int n)
   {
      try
      {
         // instanciamos dinamicamente la clase impleList
         List lst = (List)impleList.newInstance();

         // le agregamos n elementos enteros
         for( int i=0; i<n; i++ ){ lst.add(i); }

         // retornamos la lista generada
         return lst;
      }
      catch(Exception ex)
      {
         ex.printStackTrace();
         throw new RuntimeException(ex);
      }
   }
}
```

La salida de este programa, en una computadora Intel Core 2 Duo 2.1 GHz, fue la siguiente:

```
0 milisegundos (0 minutos, 0 segundos)
12297 milisegundos (0 minutos, 12 segundos)
```

Es decir: la implementación de `ArrayList` nos permitió acceder directamente a cada uno de los `n` elementos de la colección en menos de 1 milisegundo. En cambio, la implementación de `LinkedList` empleó 12 segundos para desarrollar la misma tarea.

El método `generarLista` es un *factory method* que recibe una instancia de la clase `Class`, crea dinámicamente un objeto de esa clase (que debe ser una implementación de `List`), le agrega `n` valores enteros y la retorna.

15.2.5.2 Eliminar un elemento de la colección

Como ya sabemos, los *array* son estáticos. Aunque eliminemos uno, algunos o todos sus elementos, el *array* continuará utilizando la misma cantidad de memoria.

Recordemos los conceptos de “capacidad” y “longitud”. Cuando hablamos de “capacidad” nos referimos al espacio físico del que dispone el *array* para almacenar elementos. En cambio, cuando hablamos de “longitud” nos referimos a cuánto de ese espacio físico estamos utilizando en un momento determinado.

Eliminar un elemento de un *array* puede demandarle al procesador mayor o menor esfuerzo dependiendo de la posición en la que el elemento esté ubicado. Por ejemplo, eliminar el último elemento de un *array* es tan simple como disminuir su longitud. En cambio, eliminar el primer elemento del *array* implica desplazar a todos los otros elementos una posición hacia la izquierda.

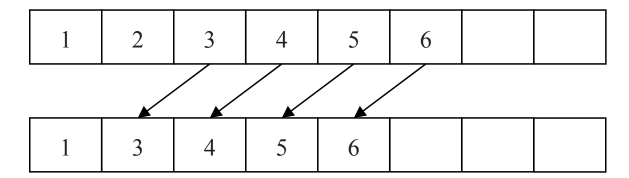


Fig. 15.3 Elimina un elemento de un *array*.

En la figura vemos que para eliminar el elemento 2 tenemos que desplazar hacia la izquierda a los elementos 3, 4, 5 y 6, lo que implica, en este caso, 4 instrucciones.

Genéricamente hablando, eliminar el *i-ésimo* elemento de un *array* implica *len-i*+1 instrucciones, siendo *len* la longitud del *array* e *i* la posición (comenzando desde cero) del elemento que queremos eliminar.

Lo anterior demuestra que el rendimiento del algoritmo decaerá cuanto mayor sea la longitud del *array* y menor sea la posición del elemento que vamos a eliminar.

En cambio, eliminar un elemento sobre una lista enlazada o doblemente enlazada siempre requiere la misma cantidad de operaciones.

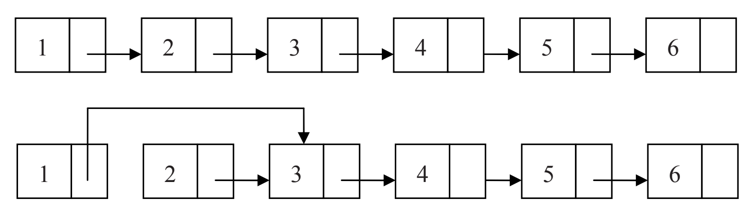


Fig. 15.4 Elimina un elemento de una lista enlazada.

En este caso, resultará mucho más eficiente utilizar una `LinkedList` en lugar de un `ArrayList`. Lo probaremos con el siguiente programa.

```
public class ArrayListVsLinkedList2
{
   public static void main(String[] args)
   {
      // generamos las dos listas
      int n = 100000;
      List arrayList = generarLista(ArrayList.class, n);
      List linkedList = generarLista(LinkedList.class, n);

      // eliminamos el primer elemento hasta que quede vacia
      Performance p1 = new Performance();
      while( !arrayList.isEmpty() ){ arrayList.remove(0); }
      System.out.println(p1);

      // eliminamos el primer elemento hasta que quede vacia
      Performance p2 = new Performance();
      while( !linkedList.isEmpty() ){ linkedList.remove(0); }
      System.out.println(p2);
   }

   // :
   // metodo generarLista...
   // :
}
```

En este caso, en la misma computadora antes referenciada, la salida fue:

```
4485 milisegundos (0 minutos, 4 segundos)
15 milisegundos (0 minutos, 0 segundos)
```

La implementación de `ArrayList` hizo que el programa demorara más de 4 segundos mientras que la implementación de `LinkedList` solo tardó 15 milisegundos.

15.2.5.3 Insertar un elemento en la colección

El análisis es análogo al anterior. Insertar un elemento en un *array* implica desplazar hacia la derecha los elementos posteriores, es decir, *len-i* desplazamientos. Además, si luego de insertar resulta colmada la capacidad del *array*, tendremos que redimensionarlo y copiar uno a uno sus elementos al nuevo espacio de memoria.

En cambio, insertar un elemento en una lista enlazada o doblemente enlazada es tan simple como crear un nuevo nodo y asignar adecuadamente las referencias `sig` y `ant`. Aquí también resultará más óptimo utilizar `LinkedList` en lugar de `ArrayList`.

15.2.6 Pilas y colas

Java provee la clase `Stack` cuya funcionalidad es la de una pila. En cambio, las colas se manejan con implementaciones de la *interface* `Queue`. `LinkedList`, por ejemplo, implementa esta *interface*.

Veamos un ejemplo que muestra cómo utilizar la clase `Stack` con sus métodos `push` (apilar), `pop` (desapilar) e `isEmpty` (está vacío).

```
package libro.cap15.stack;

import java.util.Stack;

public class TestStack
{
   public static void main(String[] args)
   {
      Stack<String> pila = new Stack<String>();
      pila.push("uno");
      pila.push("dos");
      pila.push("tres");

      while( !pila.isEmpty() )
      {
         System.out.println(pila.pop() );
      }
   }
}
```

Veamos un ejemplo que muestra cómo utilizar la *interface* `Queue` que define los métodos `add` (encolar), `poll` (desencolar) e `isEmpty` (está vacío)

```
package libro.cap15.queue;

import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;
```

```
public class TestQueue
{
   public static void main(String[] args)
   {
      Queue<String> cola = new LinkedList<String>();
      cola.add("uno");
      cola.add("dos");
      cola.add("tres");

      System.out.println( cola.poll() );
      System.out.println( cola.poll() );

      cola.add("cuatro");
      cola.add("cinco");
      cola.add("seis");

      while( !cola.isEmpty() )
      {
         System.out.println( cola.poll() );
      }
   }
}
```

■

15.3 Mapas (implementaciones de `Map`)

Llamamos "mapa" a un objeto que vincula una clave (*key*) a un determinado valor (*value*). En Java, hablar de "mapas" es hablar de implementaciones de la *interface* `Map`.

A continuación, analizaremos la clase `Hashtable`, que es una de las implementaciones de la *interface* `java.util.Map`.

15.3.1 Tablas de dispersión (`Hashtable`)

La tabla de dispersión, "tabla de hash" o *hashtable* es, tal vez, una de las estructuras de datos más flexibles.

Desde el punto de vista del usuario (programador que la utiliza), la *hashtable* es una especie de *array* que permite relacionar un valor (*value*) a una clave (*key*), ambos de cualquier tipo de datos.

Por ejemplo:

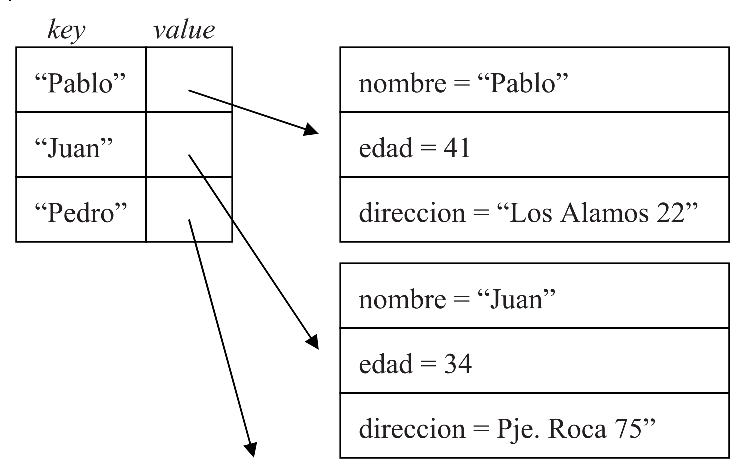


Fig. 15.5 *hashtable* que relaciona nombres (cadenas) con instancias de la clase `Persona`.

Aquí utilizamos una *hashtable* para establecer una relación unívoca entre cada instancia de la clase `Persona` y el nombre de dicha persona como clave.

Veamos primero el código de la clase `Persona` y luego un programa que instancia personas y las agrega en una *hashtable*.

```
package libro.cap15.hash;

public class Persona
{
   private String nombre;
   private int edad;
   private String direccion;

   public Persona(String nombre, int edad, String direccion)
   {
      this.nombre = nombre;
      this.edad = edad;
      this.direccion = direccion;
   }

   public String toString()
   {
      return nombre+", "+edad+" anios, vive en: "+direccion;
   }

   // :
   // setters y getters
   // :
}
```

En el siguiente programa, instanciamos personas y las agregamos en una *hashtable* utilizando sus nombres como clave. Luego le pedimos al usuario que ingrese un nombre por teclado. Este valor lo utilizaremos para acceder a la tabla, obtener la instancia de `Persona` relacionada y mostrar sus datos por pantalla.

```
package libro.cap15.hash;

import java.util.Hashtable;
import java.util.Scanner;

public class TestHashtable
{
   public static void main(String[] args)
   {
      // instanciamos una tabla que asocia personas a cadenas
      Hashtable<String, Persona> tabla = new Hashtable<String, Persona>();

      // agregamos las instancias de persona relacionandolas a las claves
      tabla.put("Pablo", new Persona("Pablo",41,"Los Alamos 22"));
      tabla.put("Juan", new Persona("Juan",34,"Pje. Roca 34"));
      tabla.put("Pedro", new Persona("Pedro",26,"San Martin 415"));

      // le pedimos al usuario que ingrese un nombre por teclado
      Scanner scanner = new Scanner(System.in);
      System.out.print("Ingrese un nombre: ");
      String nom = scanner.nextLine();

      // obtenemos el objeto relacionado al nombre ingresado y lo mostramos
      Persona p = tabla.get(nom);

      if( p!=null )
      {
         System.out.println(p);
      }
      else
      {
         System.out.println("No hay datos de "+p);
      }
   }
}
```

15.3.2 Iterar una hashtable

En el siguiente programa, instanciamos una *hashtable* y luego la recorremos secuencialmente mostrando su contenido.

```
package libro.cap15.hash;

import java.util.Enumeration;
import java.util.Hashtable;

public class TestHashtable1
{
   public static void main(String[] args)
   {
      // instanciamos una tabla con Integer (o int) como clave
      // y cadenas como value
      Hashtable<Integer, String> tabla = new Hashtable<Integer, String>();
```

```
        // le ingresamos datos
        tabla.put(1,"uno");
        tabla.put(2,"dos");
        tabla.put(3,"tres");

        // la recorremos
        for(Enumeration<Integer> e=tabla.keys(); e.hasMoreElements();)
        {
           // obtenemos la proxima clave
           int key = e.nextElement();

           // obtenemos el elemento asociado a la clave>
           String value = tabla.get(key);

           // lo mostramos por pantalla
           System.out.println(value);
        }
    }
}
```

Contrariamente a lo que se esperaría de este programa, la salida será:

```
tres
dos
uno
```

El orden de iteración no coincide con el orden en el que agregamos los elementos. Esto puede ser importante o no dependiendo de lo que necesitemos hacer.

15.3.3 Iterar una hashtable respetando el orden en que se agregaron los datos

Para solucionar el problema anterior haremos lo siguiente: crearemos una clase con una *hashtable* y un *arraylist* como variables de instancia. La llamaremos `SortedHashtable`.

Esta clase tendrá un método `put` imitando al método `put` de `Hashtable`. En este método agregaremos la *key* al *arraylist* y luego agregaremos el *value* asociado a la *key* en la *hashtable*. Veamos.

```
package libro.cap15.hash;

import java.util.ArrayList;
import java.util.Hashtable;

public class SortedHashtable<K,V>
{
   private Hashtable<K, V> table = null;
   private ArrayList<K> keys = null;

   public SortedHashtable()
   {
      table = new Hashtable<K, V>();
      keys = new ArrayList<K>();
   }
```

```
   public void put(K key, V value)
   {
      keys.add(key);
      table.put(key, value);
   }

   // sigue...
   // :
```

Luego, lo más simple será definir los métodos `keyCount`, `getKeyAt` y `get`. El primero retornará la cantidad de claves que hayamos agregado en el *arraylist* o en la *hashtable*. El segundo retornará la *i-ésima* clave agregada que coincidirá con el elemento ubicado en la misma posición del *arraylist*. El último simplemente retornará lo que retorne el método `get` de la *hashtable* de instancia.

```
   // :
   // viene de mas arriba...

   public int keyCount()
   {
      return keys.size();
   }

   public K getKeyAt(int i)
   {
      return keys.get(i);
   }

   public V get(K key)
   {
      return table.get(key);
   }
}
```

Ahora podemos replantear el programa anterior usando una instancia de nuestra clase `SortedHashtable`.

```
package libro.cap15.hash;

import java.util.Hashtable;

public class TestSortedHashtable
{
   public static void main(String[] args)
   {
      SortedHashtable<Integer, String> tabla = new SortedHashtable<Integer, String>();

      tabla.put(1,"uno");
      tabla.put(2,"dos");
      tabla.put(3,"tres");
```

```
            for(int i=0; i<tabla.keyCount(); i++)
            {
               int key = tabla.getKeyAt(i);
               String value = tabla.get(key);

               System.out.println(value);
            }
         }
      }
```

Si además queremos permitir que el usuario elimine elementos, tendremos que agregar un método `remove` que reciba la clave que se desea eliminar. Luego la eliminamos del *arraylist* y de la *hashtable*.

15.4 Estructuras de datos combinadas

La *hashtable* permite relacionar un único elemento a cada una de sus claves. Sin embargo, el hecho de que el tipo de datos del elemento sea `Object` o genérico nos habilita a crear estructuras flexibles como "*hashtables* de *arraylists*" o "*hashtables* de *hashtables* de *arraylists*", etcétera.

Veamos cómo quedaría una *hashtable* con un *arraylist* asociado a cada una de sus claves.

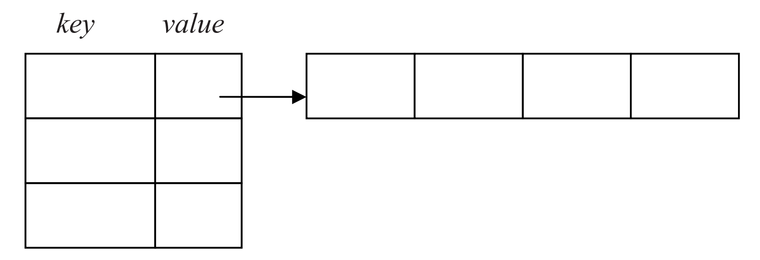


Fig. 15.6 Estructura *hashtable* de ***arraylists***.

En la siguiente línea de código declaramos un objeto `t` cuyo tipo de datos es una *hashtable* que usa cadenas como clave y *values* de tipo `ArrayList<Integer>`.

```
// declaramos una hashtable de arraylists de enteros
Hashtable<String,ArrayList<Integer> t;
```

Veamos ahora cómo quedaría una *hashtable* de *hashtables* de *arraylists*.

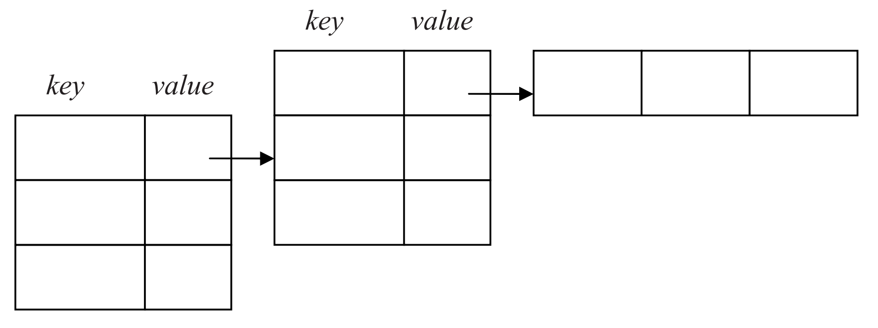


Fig. 15.7 *jashtable* de hashtables de ***arraylists***.

En la siguiente línea de código declaramos un objeto cuyo tipo es una tabla con claves de tipo `String` y *values* tipo `Hashtable<Integer,ArrayList<Persona>>`.

```
// una table de tablas de arraylists de personas
Hashtable<String,Hashtable<Integer,ArrayList<Persona>>> t;
```

15.4.1 Ejemplo de una situación real

Analizaremos un problema de la vida real donde el uso de una estructura de datos combinada como las que acabamos de ver nos facilitará enormemente la tarea.

Supongamos que nos piden desarrollar un sistema para administrar la entrega de platos y bebidas en las diferentes mesas de un restaurante. Cada mesa se identifica con un número entero (el número de mesa) y cada plato o bebida se identifica con un código numérico entero (el código de plato o bebida). Durante su permanencia, los comensales solicitan al mozo que les sirva diferentes cantidades de platos y bebidas. Luego, al finalizar el almuerzo o la cena, le solicitarán la cuenta, que consiste en un detalle de lo que han consumido y el importe total que deben pagar.

Para simplificar el análisis de este ejemplo, primero imaginemos que en el restaurante existe una única mesa, ocupada por una familia que inicialmente ordena 3 gaseosas, 2 cervezas y una tabla de quesos. Momentos más tarde, vuelven a llamar al mozo para solicitarle una gaseosa y una cerveza más. Un tiempo después piden 5 cafés y la cuenta.

Desde el punto de vista de nuestro programa tenemos que recordar todo lo que el mozo fue sirviendo en la mesa (según el ejemplo, 4 gaseosas, 3 cervezas, 1 tabla de quesos y 5 cafés). Observemos que sumamos las cantidades de las gaseosas y las cervezas ya que en un primer momento se entregaron 3 gaseosas y 2 cervezas y luego se pidieron más.

Dado que en este análisis inicial estamos suponiendo que existe una única mesa, entonces simplemente tenemos que mantener una lista de los platos y bebidas que fueron entregados y sus correspondientes cantidades.

Comenzaremos definiendo una clase `Servicio` para representar a cada producto (plato o bebida) que se entregó en la mesa, la cantidad entregada y el precio unitario.

```
package libro.cap15.ejemplo;

public class Servicio
{
   private int codigo;
   private int cantidad;
   private double precio;

   // :
   // setters y getters
   // :
}
```
■

Podríamos pensar en utilizar un *arraylist* de servicios ya que esta estructura nos permitirá mantener una lista de cada uno de los platos o bebidas entregados con sus correspondientes cantidades y precios unitarios.

Sin embargo, como existe la posibilidad de que un mismo producto se solicite más de una vez (el caso de las gaseosas y las cervezas por ejemplo) nos veremos en la situación de tener que recorrer el *arraylist* para determinar si el producto que nos están solicitando ya fue entregado con anterioridad. De ser así, solo tendremos que incrementar la cantidad. En cambio, si es la primera vez que en la mesa solicitan ese producto tendremos que agregar una nueva instancia de `Servicio` al *arraylist*.

Podemos evitar esta tediosa tarea de tener que recorrer la lista de los servicios entregados si reemplazamos el *arraylist* por una *hashtable*. En este caso la clave será el código de plato o bebida y el *value* será una instancia de `Servicio`. Así, cada vez que entregamos un servicio utilizamos el código de plato o bebida para acceder directamente al objeto que representa las cantidades entregadas de ese producto.

```
// declaramos una hashtable de codigos de platos/bebida y servicios
Hashtable<Integer,Servicio> servicios;
```

El análisis anterior lo simplificamos al suponer que en el restaurante existía una única mesa. Sin embargo, en el enunciado original se describe la existencia de varias mesas, cada una identificada por un número de mesa. Esto nos lleva a pensar en “multiplicar” la estructura anterior teniendo una “tabla de servicios” por cada mesa.

En otras palabras, necesitaremos una *hashtable* cuya clave será el número de mesa y, relacionada a cada mesa, la tabla de servicios analizada más arriba.

```
// tabla de mesas donde cada elemento es una tabla de servicios
Hashtable<Integer,Hashtable<Servicio>> mesas;
```

Supongamos ahora que el mozo nos informa que servirá una gaseosa (código 142) en la mesa número 20. Esta operación la registraremos en la estructura de datos de la siguiente manera:

```
// :
// ingresamos el numero de mesa
System.out.print("Ingrese nro. de mesa: ");
int nroMesa = scanner.nextInt();

// ingresamos el codigo de plato o bebida
System.out.print("Ingrese cod. de plato/bebida: ");
int cod = scanner.nextInt();

// ingresamos la cantidad
System.out.print("Ingrese cantidad: ");
int cant = scanner.nextInt();

// accedemos a los servicios de la mesa especificada
Hashtable<Servicio> servicios = mesas.get(nroMesa);

// pedimos el servicio que corresponde al plato/bebida especificado
Servicio s = servicios.get(cod);

// si es null es porque este plato/bebida no lo pidieron antes
if( s==null )
{
   s = new Servicio();
   s.setCodigo(cod);
   s.setCantidad(0);

   // ingresamos el precio del plato o bebida
   System.out.print("Ingrese precio unitario: ");
   double precio = scanner.nextDouble();
   s.setPrecio(precio);
```

```
   // agregamos la instancia de Servicio a la tabla
   servicios.put(cod,s);
}

// ahora si debe existir en la tabla de servicios
// un objeto servicio que represente al plato/bebida
// entonces le sumamos la cantidad solicitada
s.setCantidad( servicios.getCantidad()+cant );
// :
```

Recordemos que en Java todos los objetos son punteros; por lo tanto, `s` es un puntero a la instancia de `Servicio` almacenada en la tabla de servicios.

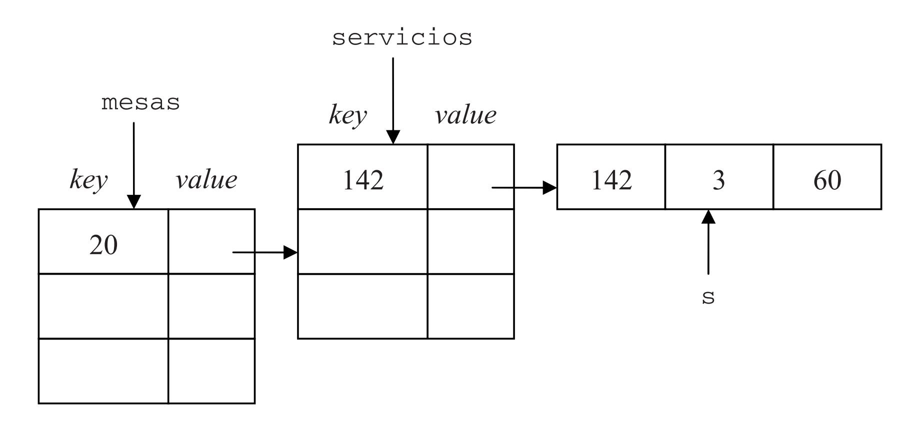


Fig. 15.8 Estructura de datos.

La estructura que utilizamos para resolver el problema es óptima ya que nos garantiza el acceso directo a los servicios entregados en cada mesa y nos permite también acceder directamente al objeto que representa a cada plato o bebida dentro de esa lista servicios.

Sin embargo, podríamos mejorarla creando una clase `Servicios` de la siguiente manera:

```
package libro.cap15.ejemplo;

public class Servicios extends Hashtable<Integer,Servicio>
{
}
```

Ahora una instancia de `Servicios` es una *hashtable* de objetos de tipo `Servicio` relacionados a claves enteras que serán los códigos de platos o bebidas.

Análogamente, podemos pensar en la clase `Mesas` de la siguiente manera:

```
package libro.cap15.ejemplo;

public class Mesas extends Hashtable<Integer,Servicios>
{
}
```

Ahora, una instancia de `Mesas` representa una tabla que vincula enteros (números de mesas) con instancias de `Servicios` donde cada una de estas es una tabla que vincula enteros (códigos de platos o bebidas) con instancias de `Servicio` que, básicamente, representan la cantidad de ese plato o bebida que ha sido entregada en una mesa y su precio unitario.

La declaración de la variable `mesas` ahora será la siguiente.

Ahora	Antes

`// ahora es`
`// mas simple :O)`
`Mesas mesas;`
	`// tabla de mesas donde cada elemento`
	`// es una tabla de servicios`
	`Hashtable<Integer,Hashtable<Servicio>> mesas;`

Con esto, podemos replantear el fragmento de código que analizamos más arriba:

```
// :
// ingresamos el numero de mesa
System.out.print("Ingrese nro. de mesa: ");
int nroMesa = scanner.nextInt();

// ingresamos el codigo de plato o bebida
System.out.print("Ingrese cod. de plato/bebida: ");
int cod = scanner.nextInt();

// ingresamos la cantidad
System.out.print("Ingrese cantidad: ");
int cant = scanner.nextInt();

// accedemos a los servicios de la mesa especificada
Servicios servicios = mesas.get(nroMesa);

// pedimos el servicio que corresponde al plato/bebida especificado
Servicio s = servicios.get(cod);

// si es null es porque este plato/bebida no lo pidieron antes
if( s==null )
{
   s = new Servicio();
   s.setCodigo(cod);
   s.setCantidad(0);

   // ingresamos el precio del plato o bebida
   System.out.print("Ingrese precio unitario: ");
   double precio = scanner.nextDouble();
   s.setPrecio(precio);

   servicios.put(cod,s);
}

// ahora si debe existir un objeto servicio representando al plato/bebida
// entonces le sumamos la cantidad solicitada
s.setCantidad( servicios.getCantidad()+cant );
```

■

15.5 Resumen

En este capítulo estudiamos las implementaciones que Java provee para trabajar con pilas, colas, listas, etcétera. También estudiamos la *hashtable* y vimos que combinando estas estructuras podemos crear soportes de datos extremadamente flexibles.

Más adelante, analizaremos estructuras de datos no lineales y desarrollaremos algoritmos complejos para analizarlos desde el punto de vista de su rendimiento. Pero antes, trabajaremos sobre un ejercicio integrador que nos permitirá aplicar todos los conocimientos que hemos venido incorporando.

15.6 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

15.6.1 Mapa conceptual

15.6.2 Autoevaluaciones

15.6.3 Presentaciones*

16

Compresión de archivos mediante el algoritmo de Huffman

Contenido

Objetivos del capítulo

- Estudiar y analizar el algoritmo de Huffman.
- Desarrollar una aplicación que, basándose en este algoritmo, permita comprimir y descomprimir archivos.
- Proveer implementaciones concretas para las *interfaces* de los objetos que se utilizan en el programa. Estas implementaciones estarán a cargo del lector y constituyen el ejercicio en sí mismo.
- Aprender sobre como manejar archivos en Java.

Competencias específicas

- Implementar aplicaciones orientadas a objetos que creen y manipulen archivos para guardar y recuperar información.



En la Web de apoyo del libro encontrará el contenido de este capítulo para leer online:

1. Ir a la pagina http://virtual.alfaomega.com.mx
2. Ingresar los nombres de Usuario y Contraseña definidos cuando registro el libro (ver Registro en la Web de apoyo).
3. Hacer un clic en el hipervínculo que lleva el nombre del capítulo que desea leer.

Módulo 4 / **Conceptos avanzados**

17

Recursividad

Contenido

Objetivos del capítulo

- Comprender el concepto de recursividad.
- Conocer el *stack* o "pila de llamadas".
- Contrastar las implementaciones recursivas e iterativas de una misma función para sacar conclusiones respecto al rendimiento y a la legibilidad del código.
- Comprender que la recursividad es una manera de dividir un problema complejo en problemas más simples y pequeños.
- Analizar el rendimiento de la implementación recursiva de función de Fibonacci.

Competencias específicas

- Comprender y aplicar la recursividad como herramienta de programación para el manejo de las estructuras de datos.

17.1 Introducción

Hasta aquí hemos desarrollado algoritmos implementados como funciones que se invocan unas a otras para dividir la tarea. Como estudiamos al inicio del libro, una tarea difícil puede dividirse en varias tareas más simples, cada una de las cuales puede a su vez dividirse en tareas más simples todavía hasta llegar a un nivel de simplicidad en el que ya no se justifique la necesidad de volver a dividir.

Sin embargo, la naturaleza de cierto tipo problemas nos inducirá a pensar soluciones basadas en funciones que se llamen a sí mismas para resolverlos. Esto no quiere decir que no puedan resolverse de "manera tradicional"; solo que, dada su naturaleza, será mucho más fácil encontrar y programar una solución recursiva que una solución iterativa tradicional.

Cuando una función se invoca a sí misma decimos que es una función recursiva.

17.2 Conceptos iniciales

17.2.1 Funciones recursivas

Una definición es recursiva cuando "define en función de sí misma". Análogamente, diremos que una función es recursiva cuando, para resolver un problema, se invoca a sí misma una y otra vez hasta que el problema queda resuelto.

En matemática encontramos varios casos de funciones recursivas. El caso típico para estudiar este tema es el de la función *factorial*, que se define de la siguiente manera:

Sea *x* perteneciente al conjunto de los números naturales (incluyendo al cero), entonces:

- *factorial*(*x*) = *x***factorial*(*x*-1), para todo *x*>0
- *factorial*(*x*) = 1, si *x*=0

La definición de la función *factorial* recurre a sí misma para expresar lo que necesita definir por lo tanto se trata de una definición recursiva.

Ahora apliquemos la definición de la función para calcular el factorial de 5. Para interpretar la tabla que veremos a continuación debemos leer la columna de la izquierda desde arriba hacia abajo y luego la columna de la derecha desde abajo hacia arriba.

factorial(5) = 5**factorial*(4) =	*factorial*(5) = 5*24 = **120**;
factorial(4) = 4**factorial*(3) =	*factorial*(4) = 4*6 = **24**;
factorial(3) = 3**factorial*(2) =	*factorial*(3) = 3*2 = **6**;
factorial(2) = 2**factorial*(1) =	*factorial*(2) = 2*1 = **2**;
factorial(1) = 1**factorial*(0) =	*factorial*(1) = 1*1 = **1**;
factorial(0) = **1**;	

Fig. 17.1 Invocaciones recursivas a la función factorial.

Como vemos, factorial de 5 se resuelve invocando a factorial de 4, pero factorial de 4 se resuelve invocando a factorial de 3, que se resuelve invocando a factorial de 2. Este se resuelve invocando a factorial de 1, que se resuelve invocando a factorial de 0 que, por definición, es 1. Luego de obtener este valor concreto podemos reemplazar todas las invocaciones que quedaron pendientes. Esto lo hacemos en la columna de la derecha, la cual, como ya dijimos, debe leerse de abajo hacia arriba.

Exponiéndolo así, el lector podría pensar que se trata de un tema complicado. Sin embargo, como veremos a continuación, el desarrollo de una función recursiva como `factorial` resulta extremadamente fácil ya que solo tenemos que ajustarnos a su definición matemática.

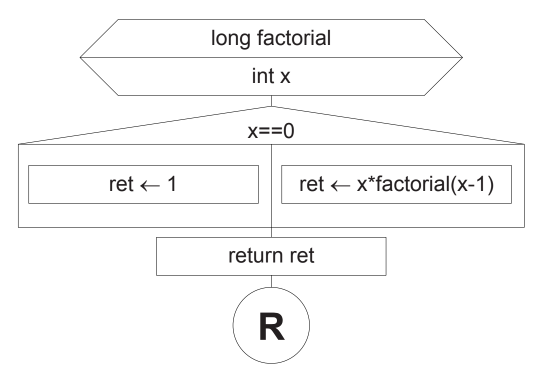


Fig. 17.2 Algoritmo recursivo de la función factorial.

La función recibe el parámetro `x`. Si `x` es igual a 0 entonces retorna 1; en cambio, si `x` es mayor que cero entonces retorna el producto de `x`, por lo que "retorna la misma función" al ser invocada con el argumento `x-1`.

17.2.2 Finalización de la recursión

Todo algoritmo recursivo debe finalizar en algún momento, de lo contrario el programa hará que se desborde la pila de llamadas y finalizará abruptamente.

En el caso de la función `factorial` la finalización de la recursión está dada por el caso particular de x=0. En este caso la función no necesita llamarse a sí misma ya que, por definición, el factorial de 0 es 1.

17.2.3 Invocación a funciones recursivas

Desde el punto de vista del programa, la función recursiva es una función común y corriente que puede invocarse como se invoca a cualquier otra función.

A continuación, veremos un programa que calcula y muestra el factorial *n*, siendo *n* un valor ingresado por el usuario.

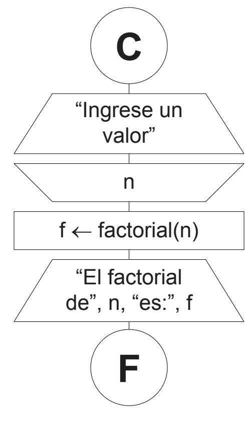


Fig. 17.3 Muestra el factorial de un valor numérico.

Veamos la codificación en lenguaje C del programa principal y la función `factorial`.

```
#include <stdio.h>

long factorial(int x);

int main()
{
   int n;
   long f;

   // pedimos un valor al usuario
   printf("Ingrese un valor: ");
   scanf("%d",&n);

   // invocamos a la funcion factorial
   f = factorial(n);

   // mostramos el resultado
   printf("El factorial de %d es: %ld\n",n,f);

   return 0;
}

long factorial(int x)
{
   long ret;

   if( x==0 )
   {
      ret = 1;
   }
   else
   {
      ret = x*factorial(x-1);
   }

   return ret;
}
```

En Java, el mismo programa se codifica así:

```
package libro.cap17;

import java.util.Scanner;

public class MuestraFactorial
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      // pedimos un valor al usuario
      System.out.print("Ingrese un valor: ");
      int n = scanner.nextInt();
```

```
      // invocamos al metodo factorial
      long f = factorial(n);

      // mostramos el resultado
      System.out.println("El factorial de "+n+" es: "+f);
   }

   private static long factorial(int x)
   {
      long ret;

      if( x==0 )
      {
         ret = 1;
      }
      else
      {
         ret = x*factorial(x-1);
      }

      return ret;
   }
}
```

En adelante optaré por implementar los diferentes ejemplos en uno u otro lenguaje, o en ambos según considere cuál pueda resultar más apropiado.

17.2.4 Funcionamiento de la pila de llamadas (stack)

Para entender la idea de "invocación recursiva" es muy importante comprender el funcionamiento del *stack* o "pila de llamadas", que es la sección de memoria donde las funciones almacenan los valores de sus variables locales y parámetros mientras dura su tiempo de ejecución.

Veamos una versión reducida del programa anterior, donde invocamos a la función `factorial` para calcular y mostrar el factorial de 4.

```
package libro.cap17;

public class MuestraFactorialDe4
{
   public static void main(String[] args)
   {
      // invocamos al metodo factorial
      long f = factorial(4);

      // mostramos el resultado
      System.out.println("El factorial de 4 es: "+f);
   }
}
```

Inicialmente, el *stack* está vacío. En el programa principal, declaramos la variable `f` y le asignamos el valor que retorna la función `factorial` pasándole el argumento: 4.

En este momento el *stack* se verá así:

Stack

Función	Variables y Parámetros	
main	f = ?	
factorial	x = 4	ret = 4 * ?

Fig. 17.4 Estado de la pila de llamadas luego de invocar a la función factorial.

La función `main` apiló en el *stack* a la variable `f` que aún no tiene valor porque está esperando que finalice la llamada a la función `factorial`.

Por su parte, la función `factorial` apiló el valor del parámetro `x` y la variable local `ret`. Como `x` es distinto de cero ingresará por el `else` para asignarle a `ret` el producto de `x` (4) por lo que retornará la invocación a `factorial` con el argumento `x-1`, es decir, 3.

Stack

Función	Variables y Parámetros	
main	f = ?	
factorial	x = 4	ret = 4 * ?
factorial	x = 3	ret = 3 * ?

Fig. 17.5 Estado de la pila de llamadas luego de la primera invocación recursiva.

La segunda llamada a la función `factorial` apila en el *stack* el valor del parámetro `x` (que es 3) y la variable local `ret`. Al ser `x` distinto de 0 ingresará por el `else` para asignarle a `ret` el producto de `x` (que es 3) por lo que retornará la función `factorial` al pasarle el argumento `x-1` que es igual a 2.

Stack

Función	Variables y Parámetros	
main	f = ?	
factorial	x = 4	ret = 4 * ?
factorial	x = 3	ret = 3 * ?
factorial	x = 2	ret = 2 * ?

Fig. 17.6 Estado de la pila de llamadas luego de la segunda invocación recursiva.

Ahora, con `x` igual a 2 volvemos a ingresar al `else` para invocar a `factorial` pasándole el argumento `x-1` = 1.

Stack

Función	Variables y Parámetros	
Main	f = ?	
factorial	x = 4	ret = 4 * ?
factorial	x = 3	ret = 3 * ?
factorial	x = 2	ret = 2 * ?
factorial	x = 1	ret = 1 * ?

Fig. 17.7 Estado de la pila de llamadas luego de la tercera invocación recursiva.

Nuevamente ingresamos al `else` invocando a `factorial` con el argumento `x-1` = 0.

Stack

Función	**Variables y Parámetros**	
`Main`	`f = ?`	
`factorial`	`x = 4`	`ret = 4 * ?`
`factorial`	`x = 3`	`ret = 3 * ?`
`factorial`	`x = 2`	`ret = 2 * ?`
`factorial`	`x = 1`	`ret = 1 * ?`
`factorial`	`x = 0`	`ret = 1`

Fig. 17.8 Estado de la pila de llamadas luego de la cuarta invocación recursiva.

En este caso, como `x` vale cero la función ingresará por el `if` para asignar el valor 1 a la variable `ret` y luego retornarlo. Al finalizar la quinta llamada el control volverá a la llamada anterior, donde quedaba pendiente asignarle a la variable `ret` el producto `x` por el valor de retorno de la función `factorial`.

Stack

Función	**Variables y Parámetros**	
`main`	`f = ?`	
`factorial`	`x = 4`	`ret = 4 * ?`
`factorial`	`x = 3`	`ret = 3 * ?`
`factorial`	`x = 2`	`ret = 2 * ?`
`factorial`	`x = 1`	`ret = 1 * 1 = 1`
~~`factorial`~~	~~`x = 0`~~	~~`ret = 1`~~

Fig. 17.9 Finaliza la cuarta invocación recursiva.

Ahora finalizará la cuarta llamada devolviendo 1 y retornando el control a la llamada anterior, donde quedaba pendiente asignarle a `ret` el producto de `x` por el valor de retorno de la función `factorial`.

Stack

Función	**Variables y Parámetros**	
`main`	`f = ?`	
`factorial`	`x = 4`	`ret = 4 * ?`
`factorial`	`x = 3`	`ret = 3 * ?`
`factorial`	`x = 2`	`ret = 2 * 2 = 2`
~~`factorial`~~	~~`x = 1`~~	~~`ret = 1 * 1 = 1`~~
~~`factorial`~~	~~`x = 0`~~	~~`ret = 1`~~

Fig. 17.10 Finaliza la tercera invocación recursiva.

Al finalizar esta llamada también se resuelve la llamada anterior:

Stack

Función	**Variables y Parámetros**	
`main`	`f = ?`	
`factorial`	`x = 4`	`ret = 4 * ?`
`factorial`	`x = 3`	`ret = 3 * 2 = 6`
~~`factorial`~~	~~`x = 2`~~	~~`ret = 2 * 2 = 2`~~
~~`factorial`~~	~~`x = 1`~~	~~`ret = 1 * 1 = 1`~~
~~`factorial`~~	~~`x = 0`~~	~~`ret = 1`~~

Fig. 17.11 Finaliza la segunda invocación recursiva.

Esto también permite resolver la llamada anterior:

Stack

Función	Variables y Parámetros	
main	f = ?	
factorial	x = 4	ret = 4 * 6 = 24
~~factorial~~	~~x = 3~~	~~ret = 3 * 2 = 6~~
~~factorial~~	~~x = 2~~	~~ret = 2 * 2 = 2~~
~~factorial~~	~~x = 1~~	~~ret = 1 * 1 = 1~~
~~factorial~~	~~x = 0~~	~~ret = 1~~

Fig. 17.12 Finaliza la primera invocación recursiva.

Y finalmente:

Stack

Función	Variables y Parámetros	
main	f = 24	
~~factorial~~	~~x = 4~~	~~ret = 4 * 6 = 24~~
~~factorial~~	~~x = 3~~	~~ret = 3 * 2 = 6~~
~~factorial~~	~~x = 2~~	~~ret = 2 * 2 = 2~~
~~factorial~~	~~x = 1~~	~~ret = 1 * 1 = 1~~
~~factorial~~	~~x = 0~~	~~ret = 1~~

Fig. 17.13 Finaliza la invocación a la función factorial.

17.2.5 Funciones recursivas vs. funciones iterativas

En general, para todo algoritmo recursivo podemos encontrar un algoritmo iterativo equivalente que resuelva el mismo problema sin tener que invocarse a sí mismo.

A continuación, compararemos las implementaciones recursiva e iterativa de la función `factorial`.

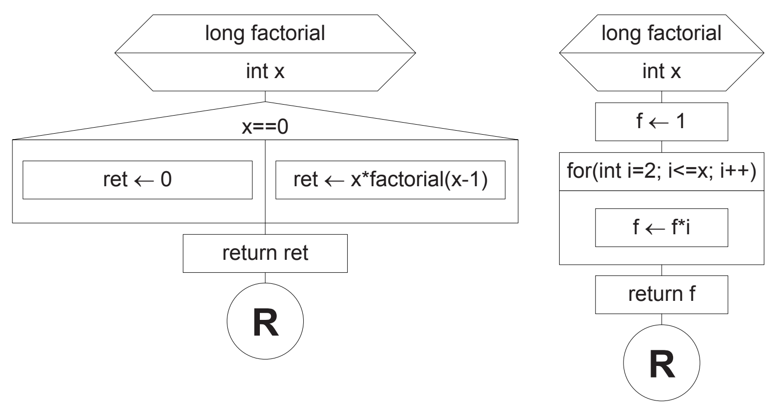


Fig. 17.14 Comparación de las soluciones recursiva e iterativa de la función factorial.

Generalmente, cuando tenemos un problema de naturaleza recursiva resulta mucho más fácil hallar una la solución que también lo sea. Sin embargo, esta solución no siempre será la más eficiente.

Para probarlo, antes de finalizar este capítulo analizaremos la función de Fibonacci que, si bien se trata de un caso particular y extremo, es un excelente ejemplo que demuestra cuán ineficiente puede llegar a resultar una implementación recursiva.

17.3 Otros ejemplos de recursividad

Veamos algunos algoritmos que pueden resolverse de forma recursiva.

Problema 17.1 - Mostrar los primeros números naturales

Mostrar por pantalla los primeros *n* números naturales, siendo *n* un valor ingresado por el usuario.

```
package libro.cap17;

import java.util.Scanner;

public class MuestraNaturales
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      int n = scanner.nextInt();
      muestraNaturales(n);
   }

   private static void muestraNaturales(int n)
   {
      if( n>0 )
      {
         // invocacion recursiva
         muestraNaturales(n-1);
      }

      System.out.println(n);
   }
}
```

Análisis

El método `muestraNaturales` recibe el valor del parámetro `n`. Si este valor es mayor que cero entonces se invoca a sí mismo con el argumento `n-1`.

En algún momento `n-1` será 0, por lo que el método no ingresará al `if` e irá directamente al `System.out.println`, donde mostrará el valor de `n` (que es 0) y retornará.

Al retornar, hará que la invocación anterior salga del `if` y llegue al `System.out.println` para mostrar el valor de `n` que, en este caso, es 1. Así sucesivamente hasta que finalicen todas las invocaciones recursivas del método `mostrarNaturales`.

Si el usuario ingresa el valor 4 entonces el programa arrojará la siguiente salida:

```
0
1
2
3
4
```

Problema 17.2 - Mostrar los primeros números naturales en orden inverso

Modificar el programa anterior para que muestre los números en orden inverso.

```
   // :
   private static void muestraNaturales(int n)
   {
      // ahora la salida la ubicamos aqui
      System.out.println(n);

      if( n>0 )
      {
         // invocacion recursiva
         muestraNaturales(n-1);
      }
   }
}
```

Análisis

El simple hecho de cambiar de lugar el `System.out.println` poniéndolo al inicio del método hará que los números se muestren en orden inverso.

Supongamos que el usuario ingresó el valor 4. Entonces, al ingresar al método, `n` vale 4. Mostramos ese valor e ingresamos al `if` para invocar al método con el argumento 3.

Ahora mostramos el valor 3 e invocamos al método con el argumento 2. Así sucesivamente hasta que `n` sea 0. En este caso el método no ingresará al `if` y finalizará, haciendo que finalicen también todas otras las llamadas recursivas.

Problema 17.3 - Recorrer recursivamente una lista enlazada

Desarrollar una implementación recursiva de la función `mostrar` analizada en el Capítulo 15. Esta función recibe un puntero a una lista enlazada y muestra el valor de cada uno de sus nodos.

```
void mostrar(Nodo* p)
{
   printf("%d\n",p->valor);

   if( p->sig!=NULL )
   {
      mostrar(p->sig);
   }
}
```

¿Qué cambios deberíamos hacer para que la función muestre la lista de atrás hacia adelante? Dejo esta tarea en manos del lector.

17.4 Permutar los caracteres de una cadena

Ahora desarrollaremos un programa que muestre por consola todas las permutaciones que se pueden realizar con los caracteres de una cadena de cualquier longitud. La cadena la ingresará el usuario por consola o por línea de comandos y no tendrá caracteres repetidos.

Por ejemplo: si la cadena ingresada fuese “ABC” entonces el programa debería arrojar las siguientes 6 líneas, en cualquier orden:

```
ABC
ACB
BAC
BCA
CAB
CBA
```

Y si la cadena fuese: “ABCD” entonces la salida debería ser:

```
ABCD
ABDC
ACBD
ACDB
ADBC
ADCB
BACD
BADC
:
```

Análisis

Evidentemente, la complejidad del problema crece conforme crece la longitud de la cadena que vamos a procesar.

Por ejemplo, si el usuario ingresa una cadena `s = "AB"` entonces la solución sería trivial ya que bastaría con mostrar `s.charAt(0)` seguido de `s.charAt(1)` y luego mostrar `s.charAt(1)` seguido de `s.charAt(0)`.

Si la cadena ingresada tuviese tres caracteres la complejidad del problema aumentaría considerablemente y, al agregar un cuarto carácter, el problema se tornaría verdaderamente difícil de resolver.

De ese modo el análisis de este ejercicio nos permitirá apreciar cómo a través de un proceso recursivo podemos reducir la complejidad de un problema hasta trivializarlo.

Veamos:

Procesar una cadena con 2 caracteres resulta ser una tarea trivial ya que, como dijimos más arriba, si la cadena fuese “AB” entonces las únicas permutaciones posibles serán “AB” y “BA”.

Para procesar una cadena con 3 caracteres debemos pensar en simplificar el problema sustrayendo un carácter. Por ejemplo, si la cadena fuese “ABC” entonces:

Sustraemos	Queda	Permutaciones	Resultado
A			
	BC		
		BC	**A**BC
		CB	**A**CB
B			
	AC		
		AC	**B**AC
		CA	**B**CA
C			
	AB		
		AB	**C**AB
		BA	**C**BA

Fig. 17.15 Permutaciones de tres caracteres.

Observemos que los datos de la columna “Resultado” surgen de concatenar el carácter sustraído con cada una de las diferentes permutaciones de los caracteres que quedan en la cadena.

Analicemos ahora el caso de una cadena con 4 caracteres como “ABCD”. Aquí, al sustraer un carácter el problema se verá un poco más simple, pero continuará siendo complicado de resolver, por lo que debemos sustraer un nuevo carácter:

Sustraemos	Queda	Sustraemos	Queda	Permutaciones	Resultado
A					
	BCD				
		B			
			CD		
				CD	**AB**CD
				DC	**AB**DC
		C			
			BD		
				BD	**AC**BD
				DB	**AC**DB
		D			
			BC		
				BC	**AD**BC
				CB	**AD**CB
B					
	ACD				
		A			
			CD		
				CD	**BA**CD
				DC	**BA**DC
		C			
			AD		
				AD	**BC**AD
				DA	**BC**DA
		D			
			AC		
				AC	**BD**AC
				CA	**BD**CA
C					
:	:	:	:	:	:

Fig. 17.16 Permutaciones de 4 caracteres.

Como se observa en esta tabla, el hecho de haber agregado el cuarto carácter nos obligó a agregar un nuevo par de columnas (“Sustraemos”, “Queda”). Es decir: si de los 4 caracteres originales sustraemos 1 entonces quedan 3. Luego, si de estos 3 caracteres sustraemos 1 quedarán 2 y estaremos en condiciones de procesarlos.

Si la cadena hubiese tenido 5 caracteres el proceso habría requerido un par de columnas (“Sustraemos”, “Queda”) adicionales, y así sucesivamente.

Cuando el razonamiento que aplicamos para resolver un determinado problema concluye diciendo “y así sucecivamente” podemos estar seguros de que estamos ante un problema de naturaleza recursiva y, por ende, su solución también lo será.

Hecho este análisis, podemos pensar en desarrollar una función recursiva que reciba como parámetro la cadena que se quiere procesar. Si esta cadena tiene 3 o más caracteres debemos sustraer uno e invocar recursivamente a la función pasándole como argumento la cadena sin el carácter sustraído. Así hasta que la cadena que se reciba como parámetro tenga solo 2 caracteres. Esta será la condición de corte de la recursión ya que estaremos en condiciones de imprimir las dos únicas permutaciones posibles.

Si la cadena que recibimos como parámetro tiene 3 o más caracteres entonces, luego de sustraer uno de ellos y antes de invocar recursivamente a la función, tenemos que concatenar el carácter sustraído en una cadena auxiliar de forma tal que, cuando llegue el momento de imprimir, podamos mostrar la cadena auxiliar compuesta por todos los caracteres sustraídos, seguida de cada una de las dos permutaciones triviales.
Veamos el código del programa:

```
package libro.cap17.permutaciones;

public class PermutacionesRec
{
   // recibe la cadena s y la cadena auxiliar (inicialmente vacia) sb
   // recordemos que sb tiene todos los caracteres sustraidos
   public static void mostrarPermutaciones(String s, StringBuffer aux)
   {
      // condicion de corte s.length()<3
      // si la cadena tiene menos de 3 caracteres entonces podemos
      // mostrar las dos permutaciones posibles
      if( s.length()<3 )
      {
         // mostramos los caracteres sustraidos seguidos de la primera permutacion
         System.out.println(aux+""+s.charAt(0)+""+s.charAt(1));

         // mostramos los caracteres sustraidos seguidos de la segunda permutacion
         System.out.println(aux+""+s.charAt(1)+""+s.charAt(0));
      }
      else
      {
         // si la cadena tiene 3 o mas caracteres...
         for(int i=0; i<s.length(); i++)
         {
            // tomamos el i-esimo caracter
            char c = s.charAt(i);

            // lo agregamos al historial de caracteres sustraidos
            aux.append(c);

            // lo eliminamos de la cadena
            String s1 = UString.removeChar(s,i);

            // llamada recursiva con un caracter menos
            mostrarPermutaciones(s1,aux);

            // removemos el caracter sustraido de la cadena auxiliar
            aux.deleteCharAt(aux.length()-1);
         }
      }
   }

   // programa principal, recibe la cadena por linea de comandos
   public static void main(String[] args)
   {
      // tomamos la cadena por linea de comandos, por ejemplo: ABCD
      String cadena = args[0];

      // la procesamos
      mostrarPermutaciones(cadena,new StringBuffer());
   }
}
```
■

Dejo planteado el siguiente interrogante para el lector: ¿Qué sucederá si modificamos el método `mostrarPermutaciones` como se muestra a continuación?

```
public static void mostrarPermutaciones(String s, StringBuffer aux)
{
   // ----- ATENCION -----
   // SOLO SE MODIFICA LA CONDICION DEL if Y EL println
   if( s.length()==1 )
   {
      // mostramos la cadena auxiliar concatenada con la cadena s
      System.out.println(aux+s);
   }
   else
   {
      // ----- ATENCION -----
      // TODO ESTO SIGUE IGUAL...

      // si la cadena tiene 3 o mas caracteres...
      for(int i=0; i<s.length(); i++)
      {
         // tomamos el i-esimo caracter
         char c = s.charAt(i);

         // lo agregamos al historial de caracteres sustraidos
         aux.append(c);

         // lo eliminamos de la cadena
         String s1 = UString.removeChar(s,i);

         // llamada recursiva con un caracter menos
         mostrarPermutaciones(s1,aux);

         // removemos el caracter sustraido de la cadena auxiliar
         aux.deleteCharAt(aux.length()-1);
      }
   }
}
```

17.5 Búsqueda binaria

El algoritmo de la búsqueda binaria puede plantearse como una función recursiva. Esta función recibirá el *array* `arr`, el valor `v` que se quiere buscar y dos índices, `i` y `j`, que delimitarán las posiciones de inicio y fin. Independientemente de qué valores contengan los índices `i` y `j`, siempre vamos a comparar a `v` con el valor que se encuentre en la posición promedio del *array* (`k=(i+j)/2`). Luego, descartaremos la mitad del *array* anterior a `k` o la mitad posterior a esta posición dependiendo de que `v` sea mayor o menor que `arr[k]`.

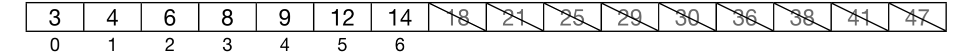
Imaginemos el siguiente *array* `arr`, ordenado ascendentemente. Las posiciones de inicio y fin son `i=0` y `j=15`, esta última coincide con `arr.length-1`.

3	4	6	8	9	12	14	18	21	25	29	30	36	38	41	47
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

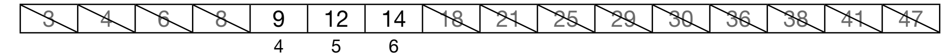
Para determinar si `arr` contiene el valor `v=14` obtenemos la posición promedio calculándola como `k=(i+j)/2`. En este caso `k` será (0+15)/2 = 7.

En la posición 7 encontraremos el valor 18 que, al ser mayor que `v`, nos permite descartar esta posición y todas las posiciones posteriores porque, al tratarse de un *array* ordenado, de estar el valor 14 estará en alguna posición anterior.

Ahora invocamos a la función recursiva pasándole el *array* `arr` y el valor que buscamos, `v`. Para descartar la mitad posterior a `k` pasaremos los índices `i=0`, `j=k-1`.



Si ahora repetimos la operación, la posición promedio será `k=3`. Como `arr[k]` es menor que `v` estamos en condiciones de descartar todas las posiciones anteriores. Para esto, invocamos a la función recursiva pasándole `arr`, `v` y los valores `i=k+1` y `j`.



La posición promedio ahora será `k=5`. Como `arr[k]` es menor que `v`, de estar el 14, debería ser en alguna posición posterior.

~~3~~	~~4~~	~~6~~	~~8~~	~~9~~	~~12~~	14	~~18~~	~~21~~	~~25~~	~~29~~	~~30~~	~~36~~	~~38~~	~~41~~	~~47~~
						6									

Luego, en la posición promedio `k=6`, `arr[k]` contiene al valor que buscábamos.

Veamos el código de la función recursiva.

```
package libro.cap17;

public class BusquedaBinariaRec
{
   public static int busquedaBinaria(int arr[], int v, int i, int j)
   {
      int k = (i+j)/2;

      // condicion de corte
      if( i>j )
      {
         return -i;
      }

      if( arr[k]==v )
      {
         return k;
      }
      else
      {
         if( arr[k]<v )
         {
            i=k+1;
         }
         else
         {
            j=k-1;
         }

         // invocacion recursiva
         return busquedaBinaria(arr, v, i, j);
      }
   }
```

```
   // test
   public static void main(String[] args)
   {
      int arr[] = {2, 4, 5, 8, 12, 18, 23, 45};
      int v=6;
      System.out.println(busquedaBinaria(arr, v, 0, arr.length-1));
   }
}
```

La condición de corte en esta función se dará cuando se crucen los índices `i` y `j`. En este caso la función retornará un valor negativo tal que, tomándolo en absoluto, coincidirá con la posición en la que debería insertarse el valor `v` que buscamos y no encontramos dentro del *array*.

17.6 Ordenamiento por selección

Una técnica para ordenar *arrays* consiste en seleccionar el elemento más pequeño y permutarlo por el que se encuentra en la primera posición, luego repetir la operación descartando la posición inicial del *array* porque esta, ahora, ya contiene su elemento definitivo.

Este algoritmo puede implementarse como una función recursiva que reciba como parámetros el *array* y una posición `dd` desde la cual debemos procesarlo. Los elementos comprendidos entre las posiciones 0 y `dd` no deben ser considerados porque estarán ordenados.

Por ejemplo, pensemos en el siguiente *array* `arr`, desordenado:

dd=0

8	5	9	3	7	4	1	6	10	2
0	1	2	3	4	5	6	7	8	9

Ahora busquemos el menor elemento entre las posiciones `dd=0` y `arr.length-1`: lo encontramos en la posición `i=6`. Luego permutamos `arr[dd]` por `arr[i]`.

dd=0 i=6

1	5	9	3	7	4	**8**	6	10	2
0	1	2	3	4	5	6	7	8	9

Ahora repetimos la operación pero descartando la primera posición del *array*. Para esto, incrementamos `dd`.

dd=1

~~1~~	5	9	3	7	4	8	6	10	2
0	1	2	3	4	5	6	7	8	9

Ahora busquemos el menor valor entre las posiciones `dd=1` y `arr.length-1`: lo encontraremos en la posición `i=9`. Los permutamos.

dd=1 i=9

~~1~~	**2**	9	3	7	4	8	6	10	**5**
0	1	2	3	4	5	6	7	8	9

Si repetimos este proceso mientras que `dd` sea menor que `arr.length` el *array* quedará ordenado.

Veamos la implementación:

```
package libro.cap17;

public class OrdenamientoSeleccionRec
{
   public static void ordenar(int arr[], int dd)
   {
      if( dd<arr.length )
      {
         // buscamos el menor valor entre dd y arr.length
         int posMin = buscarPosMinimo(arr,dd);

         // permutamos arr[dd] por arr[posMin]
         int aux=arr[dd];
         arr[dd]=arr[posMin];
         arr[posMin]=aux;

         // invocacion recursiva
         // ordenamos el array pero descartando la posicion dd
         ordenar(arr, dd+1);
      }
   }

   // buscamos el menor elemento entre dd+1 y arr.length y retornamos su posicion
   private static int buscarPosMinimo(int[] arr, int dd)
   {
      // por ahora el menor es primero...
      int posMin = dd;
      int min = arr[dd];

      for(int i=dd+1; i<arr.length; i++)
      {
         if( arr[i]<min)
         {
            min=arr[i];
            posMin=i;
         }
      }

      return posMin;
   }

   // test...
   public static void main(String[] args)
   {
      int arr[] = {3, 7, 1, 2, 9, 8, 4, 6, 5, 10};

      // ordenamos el array
      ordenar(arr, 0);

      // lo mostramos ordenado
      for(int i:arr)
      {
         System.out.println(i);
      }
   }
}
```

■



Fibonacci (c. 1170-1250) era como llamaban a Leonardo de Pisa, un matemático italiano conocido por la invención de la sucesión que lleva su nombre, que nace del estudio del aumento de la población de conejos.
En un viaje a Argelia, descubrió el sistema decimal y, consciente de sus ventajas, recorrió el Mediterráneo para aprender de los grandes matemáticos del mundo árabe.

17.7 La función de Fibonacci

La función de Fibonacci es una función recursiva que se define de la siguiente manera:

Sea *x* perteneciente al conjunto de los números naturales, entonces:

- *fibonacci*(*x*) = *fibonacci*(*x-1*) + *fibonacci*(*x-2*), para todo *x*>=3
- *fibonacci*(*x*) = 1, si *x*=1 o *x*=2

Si tomamos valores de *x* consecutivos comenzando desde 1 entonces, al invocar a *fibonacci(x)* obtendremos la sucesión de Fibonacci cuyos primeros 10 términos son los siguientes:

x	1	2	3	4	5	6	7	8	9	10	...
fibonacci(*x*)	1	1	2	3	5	8	13	21	34	55	...

Fig. 17.17 Los primeros 10 términos de la sucesión de Fibonacci.

Esta serie numérica tiene la siguiente característica: a partir de la posición 3, cada término coincide con la suma de los dos términos anteriores. Esta observación nos permite continuar la serie agregando tantos términos como queramos. Por ejemplo, el término 11 se calcula como 55+34 = 89. El término 12 será: 89+55 = 144, y así sucesivamente.

Veamos ahora las versiones recursiva e iterativa del algoritmo que resuelve esta función.

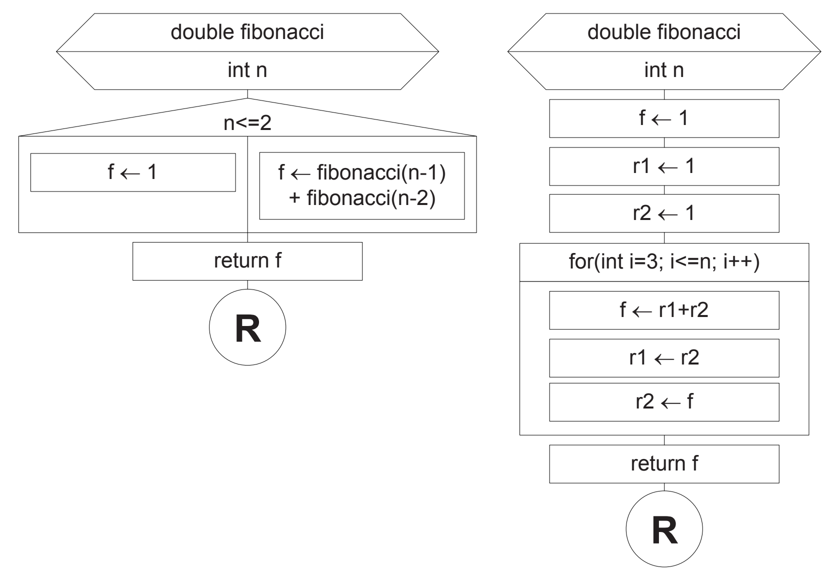


Fig. 17-18 Algoritmos recursivo e iterativo de la función de Fibonacci.

La lógica del algoritmo recursivo es trivial y no requiere explicación. Respecto del algoritmo iterativo, consiste en mantener dos variables, `r1` y `r2`, con los primeros valores de la serie que, por definición, son: *fibonacci*(2) `=` `r2` = 1 y *fibonacci*(1) = `r1` = 1. Luego, dentro del `for` calculamos `f` como la suma de `r2+r1`. Para obtener el siguiente término descartamos `r1` y consideramos que `r2` será el nuevo `r1` y que `f` será el nuevo `r2`. Así, `f` siempre tendrá la suma de los dos últimos números de la sucesión de Fibonacci.

La codificación de la implementación recursiva de la función de *fibonacci* más un programa principal la veremos a continuación.

```
package libro.cap17;

import java.util.Scanner;

public class FibRecursivo
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Ingrese un valor: ");
      int n = scanner.nextInt();

      double f = fibonacci(n);
      System.out.println("El termino " + n + " de la serie es: " + f);
   }

   private static double fibonacci(int x)
   {
      return x<=2 ? 1 : fibonacci(x-1)+fibonacci(x-2);
   }
}
```

Si el lector compila y ejecuta este programa e ingresa valores entre 1 y 30 observará que el algoritmo funciona a la perfección. En cambio, si ingresa valores entre 30 y 40 comenzará a notar cierta demora en la respuesta. Y a medida que los valores que ingrese se acerquen a 50 o se ubiquen entre 50 y 100 deberá hacerse de paciencia ya que tendrá que esperar horas, días, meses y hasta millones de años para obtener el resultado.

Perdón, ¿leí bien? ¿millones de años?

¿Por qué una computadora demoraría tanto tiempo en obtener un resultado que nosotros mismos podemos calcular en minutos? La respuesta a esta pregunta está en la doble invocación recursiva que hace la función.

Llamemos *f* a la función `fibonacci` recursiva. Para calcular *f*(50), primero debemos calcular *f*(49) y *f*(48). Claro que para calcular *f*(49) primero tenemos que calcular *f*(48) y *f*(47) y para calcular *f*(48) primero habrá que calcular *f*(47) y *f*(46), y así sucesivamente.

Para calcular *f*(*n*) siendo *n*>=3, la función se invocará a sí misma *t**2-2 veces, siendo *t* el *n-ésimo* término de la serie. Por ejemplo, *f*(31) es 1346269, entonces la función hará 2692536 llamadas recursivas para obtener el resultado. Para calcular *f*(50)=12586269025E10 la función deberá hacer 2.52E20 llamadas recursivas.

Aun así, “millones de años” suena exagerado y, obviamente, ninguno de nosotros dispone de ese tiempo para verificarlo.

Para demostrar fácilmente que no es exagerado, completaremos una tabla donde dejaremos registrado el tiempo que demora el algoritmo recursivo en resolver cada uno de los términos de la serie, particularmente los términos comprendidos entre 40 y 50.

Para medir correctamente los tiempos de ejecución utilizaremos la clase `Performance` que desarrollamos en el Capítulo 15. Repasemos su código fuente y su funcionalidad.

```java
package libro.cap17;

public class Performance
{
   private long ti; // tiempo inicial
   private long tf; // tiempo final
   private boolean stoped = false;

   public Performance()
   {
      start();
   }

   public void start()
   {
      // tomamos la fecha/hora actual, expresada en milisegundos
      ti = System.currentTimeMillis();
      stoped = false;
   }

   public void stop()
   {
      // tomamos la fecha/hora actual, expresada en milisegundos
      tf = System.currentTimeMillis();
      stoped = true;
   }

   public long getMillis()
   {
      // retorna el tiempo (en milisegundos) transcurrido entre
      // las invocaciones a los metodos start y stop
      return tf-ti;
   }

   public String toString()
   {
      if( !stoped )
      {
         stop();
      }

      long diff = tf - ti;
      long min = diff / 1000 / 60;
      long sec = (diff / 1000) % 60;

      return diff + " milisegundos (" + min + " minutos, " + sec + " segundos)";
   }

   // :
   // setters y getters
   // :
}
```

Básicamente, la clase `Performance` funciona así: en la variable `ti` (tiempo inicial) se asigna la hora, expresada en milisegundos, de inicio de la medición; y en la variable `tf` (tiempo final) se asigna la hora de finalización de la medición.

Los métodos `start` y `stop` permiten asignar la hora de inicio y la hora de fin a las variables `ti` y `tf` respectivamente. El método `getMillis` retorna la diferencia `tf-ti` que coincidirá con el tiempo transcurrido entre la invocación a los métodos `start` y `stop`.

Ahora veamos un programa que arroje los datos que nos interesa recolectar.

```
package libro.cap17;

public class FibRecursivoDe40a50
{
   private static long cont;

   public static void main(String[] args)
   {
      // con esta instancia de Performance tomaremos las mediciones
      Performance p = new Performance();

      int desde = 40; // termino inicial
      int hasta = 50; // termino final

      double ant = -1;

      for(int i=desde; i<=hasta; i++)
      {
         // inicializamos el contador
         cont=-1;

         // comenzamos a tomar el tiempo
         p.start();

         // invocamos a la funcion recursiva
         double f = fibonacci(i);

         // detenemos el tiempo
         p.stop();

         System.out.print("f("+i+") = "+f+", "+p.getMillis()+" ms, "+cont+" veces.");

         // si no es el primero entonces calculamos el incremento de tiempo
         if( ant>=0 )
         {
            System.out.println(" Incr = "+f/ant);
         }
         else
         {
            System.out.println();
         }

         // ahora, el termino anterior sera el que recien calculamos
         ant = f;
      }
   }

   private static double fibonacci(int x)
   {
      // incrementamos el contador
      cont++;
      return x<=2 ? 1 : fibonacci(x-1)+fibonacci(x-2);
   }
}
```

■

Al ejecutar este programa en mi *notebook* con procesador Intel Core 2 Duo 2.1GHz y 2 GB de memoria RAM obtuve los siguientes resultados:

```
f(40) =    1.02334155E8,   1125 ms,   204668308 veces.
f(41) =    1.65580141E8,   1906 ms,   331160280 veces. Incr = 1.618033988749895
f(42) =    2.67914296E8,   2953 ms,   535828590 veces. Incr = 1.618033988749895
f(43) =    4.33494437E8,   4813 ms,   866988872 veces. Incr = 1.618033988749895
f(44) =    7.01408733E8,   7719 ms,  1402817464 veces. Incr = 1.618033988749895
f(45) =    1.13490317E9,  12468 ms,  2269806338 veces. Incr = 1.618033988749895
f(46) =   1.836311903E9,  20203 ms,  3672623804 veces. Incr = 1.618033988749895
f(47) =   2.971215073E9,  32766 ms,  5942430144 veces. Incr = 1.618033988749895
f(48) =   4.807526976E9,  52968 ms,  9615053950 veces. Incr = 1.618033988749895
f(49) =   7.778742049E9,  86204 ms, 15557484096 veces. Incr = 1.618033988749895
f(50) = 1.2586269025E10, 139375 ms, 25172538048 veces. Incr = 1.618033988749895
```

El programa registró el tiempo que demoró la función recursiva en resolver los términos de la serie de Fibonacci comprendidos entre 40 y 50. También registró la cantidad de llamadas recursivas que realizó la función para resolver cada uno de estos términos.

Podemos observar que el paso de un término al siguiente representa un incremento de tiempo de, aproximadamente, el 61%. Esta proporción también se da en el incremento de la cantidad de llamadas recursivas.

Por ejemplo: *f*(47) demoró en resolverse 32766 milisegundos y *f*(46) demoró 20203 milisegundos. Si dividimos: 32766/20203 = 1.6180... Análogamente, *f*(47) realizó 5942430144 llamadas recursivas y *f*(46) realizó 3672623804. Si dividimos estos valores: 5942430144/3672623804 = 1.6180...



Un número áureo es un número algebraico irracional con muchas propiedades interesantes, descubierto en la antigüedad como relación o proporción entre segmentos de rectas. Dicha proporción la encontramos en en algunas figuras geométricas y en la naturaleza (en las nervaduras de las hojas de algunos árboles, en el grosor de las ramas, en el caparazón de un caracol, etc.)
Hay quien cree que posee una importancia mística.

La misma proporción se mantiene también al avanzar de un término de la serie al siguiente. Por ejemplo, *f*(4)/*f*(3) = 3/2 = 1.5, pero *f*(5)/*f*(4) = 5/3 = 1.66. Y *f*(6)/*f*(5) = 8/5 = 1.6 y *f*(7)/*f*(6) = 13/8 = 1.625 y, a medida que avancemos en los términos, veremos que el resultado tenderá a 1.61803398... que es una aproximación de: $(\sqrt{5}+1)/2$, conocido como número áureo.

Por lo tanto, conociendo el tiempo que el algoritmo demora en resolver el término número 40, podemos calcular cuánto más demorará en resolver cualquier otro término.

Por ejemplo:

$$\begin{aligned} demora(f(41)) &= demora(f(40)) * 1.618033988749895 \\ &= 1125 * 1.618033988749895 \\ &= 1820 \text{ aprox.} \end{aligned}$$

Esto nos da una diferencia de 86 milisegundos menos respecto de la medición que arrojó el programa.

Probemos con otro término:

$$\begin{aligned} demora(f(48)) &= demora(f(47)) * 1.618033988749895 \\ &= 32766 * 1.618033988749895 \\ &= 53016 \text{ aprox.} \end{aligned}$$

La diferencia aquí es de 48 milisegundos respecto de la medición.

Si partimos de la demora de *f*(50), entonces podemos calcular:

$$\begin{aligned} demora(f(51)) &= demora(f(50)) * 1.618033988749895^{1} \\ demora(f(52)) &= demora(f(50)) * 1.618033988749895^{2} \\ demora(f(53)) &= demora(f(50)) * 1.618033988749895^{3} \\ &\vdots \\ demora(f(100)) &= demora(f(50)) * 1.618033988749895^{50} = \\ &demora(f(50)) * 28143753341 = \\ &139375 * 28143753123 = \\ &3.92254E+15 \text{ milisegundos.} \end{aligned}$$

Si este valor lo dividimos por 1000 lo estaremos expresando en segundos, y si lo volvemos a dividir por 60 lo expresaremos en minutos. Luego, al dividirlo por 60 lo tendremos expresado en horas. Finalmente, si lo dividimos por 24 y luego por 365 sabremos cuántos años le tomaría a una computadora resolver el término número 100 de la serie de Fibonacci mediante el algoritmo recursivo. Veamos:

3.92254E+15 / 1000*60*60*24*365 = 1224382 años (aproximadamente).

17.7.1 Optimización del algoritmo recursivo de Fibonacci

Como vimos, la función recursiva de Fibonacci resulta tan ineficiente porque debe resolver varias veces el mismo cálculo.

Recordemos: para calcular *f*(50) primero debe resolver *f*(49) y *f*(48), pero para calcular *f*(49) primero debe resolver *f*(48) y *f*(47), y para calcular *f*(48) primero debe resolver *f*(47) y *f*(46).

Sin embargo, podemos optimizar la función "ayudándole" a recordar los términos que ya calculó y de esta manera evitar que los tenga que volver a calcular. Para esto, utilizaremos una *hashtable* en la que almacenaremos los términos de la sucesión a medida que la función los vaya calculando.

```
private static double fibonacci(int x,Hashtable<Integer,Double> t)
{
   // primero verificamos si el resultado esta en la tabla
   Double d = t.get(x);

   // si no estaba entonces lo calculamos y lo ingresamos en la tabla
   if( d==null )
   {
      d = fibonacci(x-1, t)+fibonacci(x-2, t);
      t.put(x, d);
   }

   // retornamos el resultado
   return d;
}
```
■

Notemos que la tabla debe inicializarse con las filas correspondientes a *fibonacci*(1) = *fibonacci*(2) = 1.

Con esta mejora, el rendimiento de la función recursiva es comparable al rendimiento de la implementación iterativa.

Veamos entonces un programa que muestra los primeros *n* términos de la serie de Fibonacci siendo *n* un valor que ingresa el usuario.

```
package libro.cap17;

import java.util.Hashtable;
import java.util.Scanner;

public class FibRecursivoOptimizado
{
   public static void main(String[] args)
   {
      // inicializamos la tabla
      Hashtable<Integer, Double> t = new Hashtable<Integer, Double>();
      t.put(1, 1d);  // 1d => 1 convertido a double
      t.put(2, 1d);  // 1d => 1 convertido a double
```

```
        // el usuario ingresa el valor de n
        Scanner scanner = new Scanner(System.in);
        System.out.print("Cuantos terminos quiere ver: ");
        int n = scanner.nextInt();

        for( int i=1; i<=n; i++)
        {
           double f = fibonacci(i,t);
           System.out.println("fib(" + i + ") = " + f);
        }
     }

     // :
     // aqui va la funcion...
     // :
  }
```

17.8 Resumen

El concepto de recursividad que estudiamos aquí es fundamental para estudiar el tema del próximo capítulo: árboles. Los árboles son estructuras dinámicas de naturaleza recursiva y, por lo tanto, para desplazarnos sobre sus nodos tendremos que utilizar algoritmos recursivos.

17.9 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

17.9.1 Mapa conceptual

17.9.2 Autoevaluaciones

17.9.3 Presentaciones*

18

Árboles

Contenido

Objetivos del capítulo

- Estudiar la estructura de árbol, sus características y clasificaciones.
- Analizar los árboles binarios y las diferentes maneras de recorrerlos.
- Comparar las implementaciones recursivas e iterativas de los recorridos en *pre*, *post* e *inorden*.
- Estudiar el Árbol Binario de Búsqueda (ABB).
- Estudiar los árboles *n-areos*.
- Desacoplar el procesamiento de los nodos durante el recorrido.
- Analizar y contrastar soluciones de procesamiento basadas en iteradores vs. implementación basadas en *callback methods* (retrollamadas).

Competencias específicas

- Comprender y aplicar la recursividad como herramienta de programación para el manejo de las estructuras de datos.
- Conocer, identificar y aplicar las estructuras no lineales en la solución de problemas del mundo real.

18.1 Introducción

Llamamos árbol a una estructura de datos no lineal, recursiva, en la que se observa que desde cada nodo descienden uno o varios nodos de su mismo tipo salvo los últimos, que no tienen descendencia.

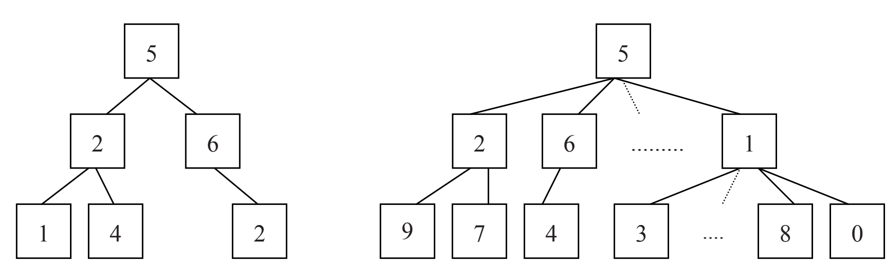


Fig. 18.1 Árbol binario.

Fig. 18.2 Árbol *n-ario*.

En la figura vemos dos árboles. En el primer caso, de cada nodo (padre) descienden hasta 2 nodos (hijos). En el segundo, la cantidad de nodos (hijos) que descienden de cada nodo (padre) es variable y está representada por los puntos suspensivos.

De cualquier manera, siempre tendremos un nodo inicial al que llamaremos "raíz" desde el cual descenderán una cantidad finita, fija o variable, de nodos de su mismo tipo, a los que llamaremos "hijos". A su vez, cada nodo hijo puede ser raíz o "padre" de otros hijos y así sucesivamente hasta llegar a las "hojas" del árbol, que no son otra cosa que nodos sin descendencia.

Los árboles pueden clasificarse en función de la cantidad hijos que desciendan de cada padre. Así, diremos que el árbol de la Fig. 18.1 es un árbol binario; en cambio, el árbol que se muestra en la Fig. 18.2 es un árbol *n-ario* ya que cada nodo puede tener una cantidad *n* de hijos, siendo *n* un valor que puede variar entre un nodo padre y otro.

Como ejemplo, pensemos en los siguientes casos:

- El organigrama de una organización.
- El sistema de archivos de Windows o Linux.
- Un sistema de ventas de tipo piramidal en el que cada vendedor vende sus productos a través de los vendedores que él mismo reclutó, cada uno de los cuales vende sus productos a través de los vendedores que, también, él mismo reclutó, y así sucesivamente.
- El árbol genealógico de una familia. ¿Es un árbol tal como lo definimos más arriba?

18.1.1 Tipos de árbol

Como dijimos más arriba, según la cantidad de hijos que cada nodo padre pueda llegar a tener diremos que el árbol es binario (2 hijos), ternario (3 hijos), cuaternario (4 hijos) o, *n-ario*, siendo *n* un valor mayor o igual a 1 y, tal vez, diferente para cada nodo padre.

18.1.2 Implementación de la estructura de datos

A continuación, veremos las estructuras que nos permitirán implementar un árbol binario y un árbol *n-ario*.

En lenguaje C:

Nodo de un árbol binario	Nodo de un árbol *n-ario*
`typedef struct Nodo { int v; struct Nodo *izq; struct Nodo *der; }Nodo;`	`typedef struct NodoN { int v; struct NodoN *hijos[]; }NodoN;`

```
typedef struct Nodo
{
   int v;
   struct Nodo *izq;
   struct Nodo *der;
}Nodo;
```

```
typedef struct NodoN
{
   int v;
   struct NodoN *hijos[];
}NodoN;
```

Como era de esperar, el nodo del árbol binario tiene dos punteros (`izq` y `der`) para referenciar a sus, a lo sumo, dos hijos. En cambio el nodo del árbol *n-ario* tiene un *array* de punteros que nos permitirá hacer referencia a una cantidad de hijos diferente para cada padre o raíz.

En lenguaje Java:

Nodo de un árbol binario

```
public class Nodo
{
   private int v;
   private Nodo izq = null;
   private Nodo der = null;

   // :
   // setters y getters
   // :
}
```

Nodo de un árbol *n-ario*

```
public class NodoN
{
   private int v;
   private ArrayList<NodoN> hijos;

   // :
   // setters y getters
   // :
}
```

Recordemos que en Java los objetos son punteros; por lo tanto, las referencias a los hijos del nodo son objetos de su mismo tipo. Claro que estos nodos podrían ser genéricos, de forma tal que permitan contener datos de cualquier tipo. Veamos las versiones, genéricas en `T`, del árbol binario y del árbol *n*-ario.

Nodo genérico de un árbol binario

```
public class Nodo<T>
{
   private T v;
   private Nodo<T> izq = null;
   private Nodo<T> der = null;

   // :
   // setters y getters
   // :
}
```

Nodo genérico de un árbol *n-ario*

```
public class NodoN<T>
{
   private T v;
   private ArrayList<NodoN<T>> hijos;

   // :
   // setters y getters
   // :
}
```

18.2 Árbol binario

El árbol binario requiere especial atención ya que es la estructura de datos que dará soporte a una gran variedad de algoritmos que estudiaremos en este libro:

- Algoritmos de búsqueda (árbol binario de búsqueda).
- Algoritmos de ordenamiento (*heapsort*).
- Algoritmos de compresión y/o encriptación (Huffman, estudiado en el Capítulo 16).

18.2.1 Niveles de un árbol binario

Dado un árbol binario, si consideramos que la raíz constituye el primer nivel del árbol (nivel 0) entonces sus nodos hijos constituirán el segundo nivel (nivel 1), sus "nietos" el tercero (nivel 2) y así sucesivamente.

18.2.2 Recorrer los nodos de un árbol binario

Recorrer un árbol implica desplazarnos a través de todos sus nodos respetando un determinado orden o criterio. Según cuál sea el orden o criterio que utilicemos para emprender la recorrida, tendremos que hablar de recorridos "en amplitud" (también conocido como "recorrido por niveles") o "en profundidad".

18.2.3 Recorrido en amplitud o "por niveles"

El recorrido "por niveles" consiste en listar los nodos del árbol desde arriba hacia abajo y de izquierda a derecha.

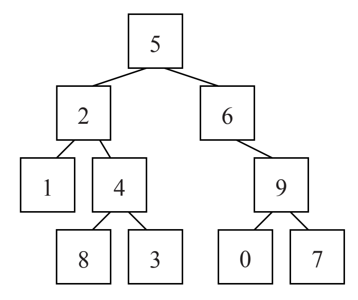


Fig. 18.3 Árbol binario.

El recorrido "por niveles" sobre este árbol binario es el siguiente: 5, 2, 6, 1, 4, 9, 8, 3, 0, 7.

Para implementar un algoritmo que nos permita emprender este recorrido tenemos que utilizar una cola, como veremos a continuación.

La estrategia es la siguiente:

- Encolamos la raíz del árbol.
- Luego, mientras la cola tenga elementos:
    - Tomamos un elemento (nodo) de la cola.
    - Procesamos el elemento (por ejemplo, mostramos su valor en la pantalla).
    - Encolamos a sus hijos.

Veamos el algoritmo en la Fig. 18.4.

18.2.4 Recorridos en profundidad (preorden, postorden e inorden)

Cada nodo de un árbol binario es en sí mismo la raíz de un subárbol binario. Luego, dependiendo del orden en que se procese el nodo raíz y cada uno de sus hijos hablaremos de recorridos *pre*, *post* e *inorden*.

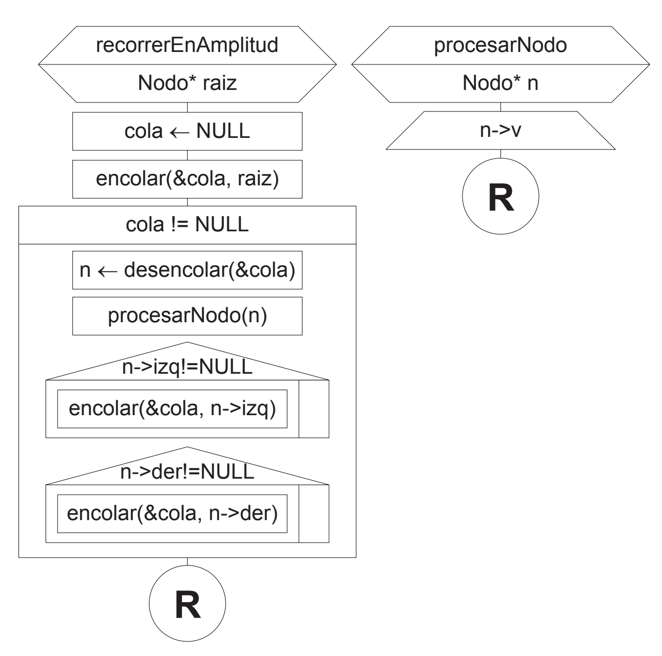


Fig. 18.4 Recorrido por niveles de los nodos de un árbol binario.

Preorden consiste en procesar primero la raíz y luego su hijo izquierdo y su hijo derecho. *Postorden* se refiere a que el proceso de la raíz será posterior al proceso de sus hijos. Por último, *inorden* u orden simétrico significa que primero se procesará al hijo izquierdo, luego la raíz y finalmente al hijo derecho.

Como cada hijo es en sí mismo la raíz de un subárbol estos recorridos, generalmente, se implementan como funciones recursivas.

Recordemos el árbol binario analizado más arriba para comparar cada uno de los recorridos en profundidad.

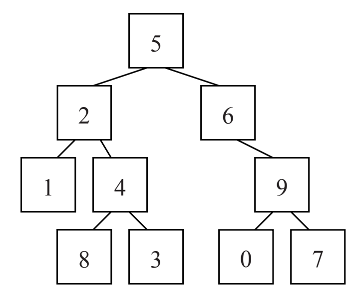


Fig. 18.5 Árbol binario.

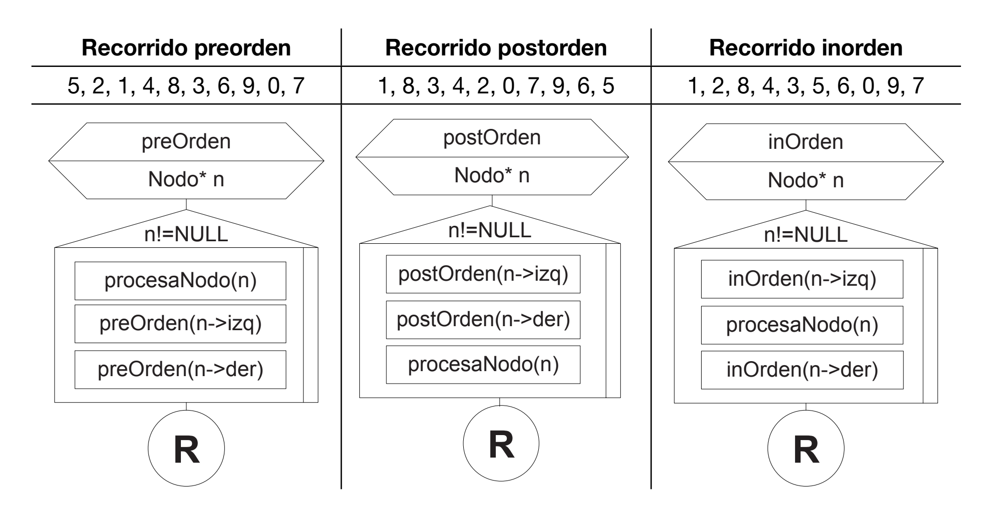


Fig. 18.6 Recorridos en profundidad.

Supongamos que `procesaNodo` simplemente imprime el valor del nodo en la consola. Entonces, al procesar los nodos del árbol binario de la figura anterior en *preorden* obtendremos la siguiente salida: 5, 2, 1, 4, 8, 3, 6, 9, 0, 7.

Esto surge del siguiente análisis: primero procesamos la raíz del árbol 5 y luego a sus hijos 2 y 6, pero 6 debe esperar a que finalice el proceso de 2. Procesamos a 2 y luego a sus hijos 1 y 4, pero 4 debe esperar a que finalice el proceso de 1. Luego procesamos a 1 que, al no tener hijos, finaliza su proceso. El proceso pendiente más reciente o el último proceso apilado es el de 4, entonces lo procesamos y luego a sus hijos, 8 y 3, aunque 3 debe esperar a que finalice el proceso de 8. El proceso de 8 finalizará inmediatamente, permitiendo que también finalice el proceso de 3, con lo que también finalizará el proceso de todo el subárbol izquierdo de 5. Ahora puede avanzar el proceso de 6 que estaba apilado. Entonces procesamos a 6 y luego a su hijo derecho (9) ya que 6 no tiene hijo izquierdo. Luego procesamos 9 y sus hijos 0 y 7, cuyos procesos serán inmediatos porque ninguno de los dos tiene hijos.

Dada la naturaleza recursiva de la estructura del árbol, la implementación recursiva de los algoritmos de recorrido resulta ser simple y natural. Sin embargo, en ocasiones puede que no sea viable utilizar una implementación recursiva, situación que nos obligará a pensar en alternativas iterativas.

18.2.5 Implementación iterativa del recorrido en preorden

La implementación iterativa del recorrido en *preorden* a través de los nodos de un árbol binario se base en el uso de una pila. La idea es apilar un nodo padre (en principio la raíz) y luego, mientras la pila no esté vacía, tomar un elemento, procesarlo y apilar sus hijos.

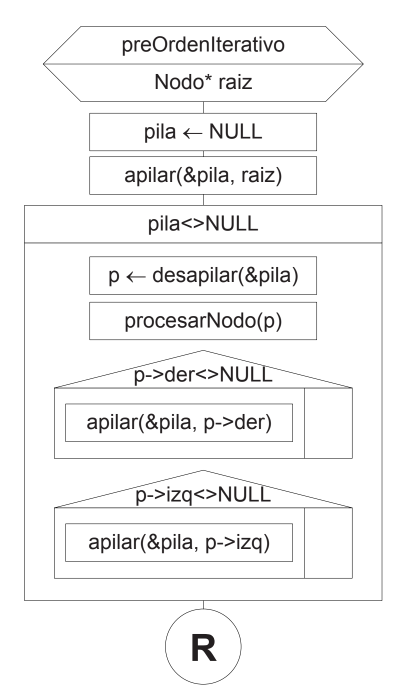


Fig. 18.7 Recorridos en profundidad, implementación iterativa.

Considerando el árbol que analizamos más arriba, entonces primero apilaremos la raíz 5. Luego, dentro del `while` tomamos un elemento (el 5), lo procesamos y apilamos su hijo derecho 6 y su hijo izquierdo 2. En la siguiente iteración, al tomar un elemento de la pila obtendremos el último elemento apilado: el 2. Luego lo procesamos y apilamos sus hijos y continuamos así mientras que la pila no quede vacía.

18.2.6 Implementación iterativa del recorrido en postorden

La versión iterativa del recorrido *postorden* no es tan simple como la anterior. Sin embargo, podemos deducirla analizando cuidadosamente el árbol binario.

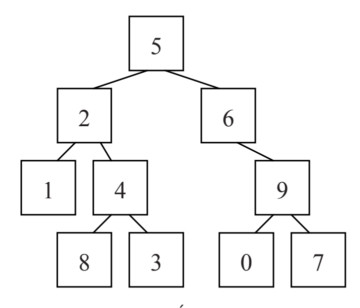


Fig. 18.8 – Árbol binario.

Recordemos que la secuencia del recorrido *postorden* sobre los nodos de este árbol binario es: 1, 8, 3, 4, 2, 0, 7, 9, 6, 5.

Aquí también nos apoyaremos en una pila. Comenzaremos apilando el nodo raíz (5) e iteraremos mientras que la pila no esté vacía. Luego, dentro del `while` desapilamos un elemento.

En la primera iteración aparecerá el 5 (la raíz). Como este nodo tiene hijos debemos darles prioridad; por lo tanto, lo volveremos a apilar y luego apilaremos a su hijo derecho y a su hijo izquierdo.

En la segunda iteración aparecerá 2, hijo izquierdo de la raíz. Como este nodo también tiene hijos debemos darles prioridad; por lo tanto, lo volveremos a apilar y lo mismo haremos con su hijo derecho y con su hijo izquierdo.

En la tercera iteración aparecerá el 1, hijo izquierdo de 2. Como este nodo no tiene hijos entonces estaremos en condiciones de procesarlo.

El próximo elemento en salir de la pila será 4, hijo derecho de 2 que, como tiene hijos, debemos darles prioridad. Entonces lo volvemos a apilar y luego apilamos su hijo derecho y a su hijo izquierdo.

En la próxima iteración saldrá el 8, hijo izquierdo de 4 que, al no tener hijos, estaremos en condiciones de procesarlo.

En la próxima iteración saldrá el 3, que también podremos procesar porque no tiene hijos.

En la próxima iteración saldrá el 4 que, si bien tiene hijos, estos ya fueron procesados, por lo que ahora sí estaremos en condiciones de procesarlo.

Todo este análisis demuestra que para poder procesar el nodo que desapilamos deberá cumplirse alguna de las siguientes condiciones:

- Que el nodo sea una hoja (es decir, que no tenga hijos).
- Que el nodo haya quedado pendiente de ser procesado porque se le dio prioridad de proceso a sus hijos.

Por lo tanto, para completar el algoritmo debemos pensar también en una lista de nodos pendientes de ser procesados.

Luego, cada nodo desapilado y vuelto a apilar también debe ser agregado a la lista de nodos pendientes. Ver Fig. 18.9.

Como dijimos más arriba, el algoritmo se resuelve apoyándose en una pila y una lista. Para simplificar la lectura utilizamos las variables booleanas `sinHijos` y `pendiente`. La primera será `true` o `false` según `p`, el nodo desapilado, tenga o no tenga hijos. La segunda será `true` si `p` ya fue agregado a la lista; de lo contrario será `false`.

Luego, si el nodo desapilado no tiene hijos o está pendiente de ser procesado entonces lo procesamos y forzamos una nueva iteración con la sentencia `continue`.

Si el nodo desapilado no cumple con la condición anterior será porque tiene hijos o porque no está pendiente de ser procesado.

Particularmente, si tiene al menos un hijo entonces debemos darle prioridad. Para esto, volvemos a apilar al nodo desapilado (padre), lo agregamos a la lista de nodos pendientes de proceso y luego apilamos su hijo derecho y su hijo izquierdo.

La implementación iterativa del recorrido *inorden* quedará a cargo del lector.

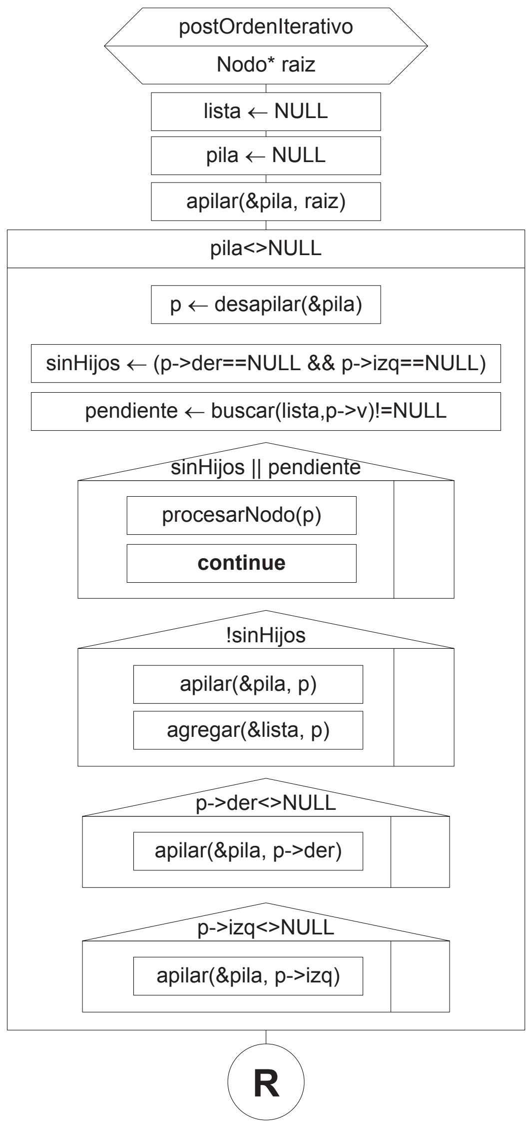


Fig. 18.9 Implementación iterativa de un recorrido postorden.



La sentencia `continue` traslada el control a la cabecera del ciclo evitando que se ejecuten las instrucciones ubicadas más abajo.
En este caso, si se verifica la proposición `sinHijos || pendiente` entonces, luego de invocar a `procesarNodo` el algoritmo volverá a la cabecera del ciclo para evaluar la condición y, si corresponde, continuar iterando.

18.3 Árbol binario en Java, objetos



Tal como lo hicimos cuando, usando clases, encapsulamos las estructuras lineales dinámicas (listas, pilas, colas), también podemos pensar en encapsular la lógica asociada a los árboles binarios.

Esta lógica, en principio, estará directamente relacionada con los tipos de recorrido que se pueden emprender a través de sus nodos (*pre*, *post* e *inorden*) y con la necesidad de desacoplar el proceso que el usuario (programador que usa nuestras clases) quiera aplicarle a cada nodo a medida que avanza en el recorrido.

Para esto, analizaremos dos enfoques distintos:

- El primero tendrá un estilo parecido a la programación estructurada.
- El segundo será más puro respecto de la programación orientada a objetos.

18.3.1 Enfoque basado en una clase utilitaria

Como estudiamos en el Capítulo 14, las clases utilitarias son aquellas que tienen todos o gran cantidad de sus métodos estáticos.

Dado que los métodos estáticos no tienen acceso a las variables de instancia este enfoque nos obligará a mantener separadas las operaciones y la estructura de datos propiamente dicha, al estilo de la programación estructurada.

Como dijimos más arriba, por el momento nos ocuparemos de encapsular los diferentes tipos de recorridos y de separar el proceso que el usuario le quiera aplicar a cada nodo durante dicho recorrido. Por lo tanto, vamos a desarrollar la clase utilitaria `UArbol` que tendrá un método para cada uno de los diferentes tipos de recorrido analizados.

Por otro lado, si emprendemos un recorrido a través de los nodos del árbol será porque los queremos procesar. Este proceso puede ser, por ejemplo, mostrar su valor por pantalla o, tal vez, algo con mayor nivel de complejidad.

Debemos abstraernos de cuál será el proceso que el usuario (programador que utilice la clase `UArbol`) quiera aplicar o ejecutar sobre cada nodo. Para esto, simplemente definiremos una *interface* con un método `ejecutar` y así será el usuario el responsable de proveer una implementación concreta de dicho método. Nosotros simplemente lo invocaremos a medida que transitemos sobre los diferentes nodos del árbol durante el recorrido.

Veamos entonces la clase `Nodo` que será genérica en `T`.

```
package libro.cap18.arbolbinario;

public class Nodo<T>
{
   // el nodo guarda un valor de tipo T (generico)
   private T v = null;

   // hijo izquierdo e hijo derecho
   private Nodo<T> izq = null;
   private Nodo<T> der = null;

   // constructor
   public Nodo(T v)
   {
      this.v = v;
   }

   // :
   // setters y getters
   // :
}
```

Ahora la *interface* `Proceso` que define el método `ejecutar`.

```
package libro.cap18.arbolbinario;

public interface Proceso<T>
{
   public void ejecutar(Nodo<T> n);
}
```

Teniendo definidos el nodo y la *interface* podemos analizar la clase `UArbol`.

18.3.2 Recorrido por niveles o "en amplitud"

Comenzaremos por programar el método (estático) `amplitud` que permitirá emprender un recorrido a través de los diferentes niveles de un árbol binario.

Recordemos que este tipo de recorrido se basa en una cola para encolar la raíz del árbol. Luego, mientras la cola tenga elementos, desencolamos un nodo, lo procesamos y si tiene hijos los encolamos también.

Observemos que el método `amplitud` recibe dos parámetros: la raíz del árbol binario (`root`) y una instancia de `Proceso` (`p`). Esto lo analizaremos luego de ver su código fuente.

```
package libro.cap18.arbolbinario;

import java.util.LinkedList;
import java.util.Queue;

public class UArbol
{
   // procesa "por niveles" los nodos del arbol binario
   public static <T> void amplitud(Nodo<T> root, Proceso<T> p)
   {
      // creamos un cola
      Queue<Nodo<T>> cola = new LinkedList<Nodo<T>>();

      // encolamos la raiz
      cola.add(root);

      // iteramos mientras que la cola no quede vacia
      while( !cola.isEmpty() )
      {
         // desencolamos un elemento
         Nodo<T> n = cola.poll();

         // lo procesamos
         p.ejecutar(n);

         // si el nodo desencolado tiene hijo izquierdo => lo encolamos
         if( n.getIzq() != null )
         {
            cola.add(n.getIzq());
         }

         // idem con el hijo derecho
         if( n.getDer() != null )
         {
            cola.add(n.getDer());
         }
      }
   }
}
```

El hecho de recibir la raíz del árbol como parámetro implica que la responsabilidad de cargar los datos en la estructura será exclusiva del usuario (programador que use `UArbol`). Nosotros solo le facilitaremos, en este caso, un método para recorrerlo "en amplitud".

Por otro lado, también será el usuario el responsable de proveer una implementación concreta de la *interface* `Proceso`. Esto nos permitirá aplicar el proceso a cada nodo sin tener que preocuparnos por la implementación.

Hasta aquí llegó nuestra tarea como programadores de la clase `UArbol`. Ahora, si nos ponemos en los zapatos del usuario (programador que la utilizará), tendremos que proveer una implementación de `Proceso` que sobrescriba adecuadamente el método `ejecutar`.

Por ejemplo, la siguiente clase implementa la *interface* `Proceso` y sobrescribe el método `ejecutar`, donde imprime por pantalla el valor del nodo que está siendo procesado.

```
package libro.cap18.arbolbinario;

public class ProcesoImpleImprime implements Proceso<Integer>
{
   public void ejecutar(Nodo<Integer> n)
   {
      // simplemente imprimimos el valor del nodo n en la consola
      System.out.println(n.getV());
   }
}
```

Ahora, veamos un programa que crea un árbol binario ("hardcodeado") y lo recorre por niveles mostrando el valor de cada uno de sus nodos.

```
package libro.cap18.arbolbinario;

public class TestArbol
{
   public static void main(String[] args)
   {
      // el metodo crearArbol "arma" un arbol binario y retorna la raiz
      Nodo<Integer> raiz = crearArbol();

      // recorremos por niveles y proceso cada nodo
      UArbol.amplitud(raiz, new ProcesoImpleImprime());
   }

   // "hardcodea" un arbol binario y retorna su raiz
   private static Nodo<Integer> crearArbol()
   {
      // nivel 3
      Nodo<Integer> n8 = new Nodo<Integer>(8);
      Nodo<Integer> n3 = new Nodo<Integer>(3);
      Nodo<Integer> n0 = new Nodo<Integer>(0);
      Nodo<Integer> n7 = new Nodo<Integer>(7);

      // nivel 2
      Nodo<Integer> n1 = new Nodo<Integer>(1);
      Nodo<Integer> n4 = new Nodo<Integer>(4);
      n4.setIzq(n8);
      n4.setDer(n3);
      Nodo<Integer> n9 = new Nodo<Integer>(9);
      n9.setIzq(n0);
      n9.setDer(n7);
```

```
      // nivel 1
      Nodo<Integer> n2 = new Nodo<Integer>(2);
      n2.setIzq(n1);
      n2.setDer(n4);
      Nodo<Integer> n6 = new Nodo<Integer>(6);
      n6.setDer(n9);

      // nivel 0
      Nodo<Integer> n5 = new Nodo<Integer>(5);
      n5.setIzq(n2);
      n5.setDer(n6);

      return n5;
   }
}
```

¿Qué cambios tendríamos que hacer si en lugar de mostrar por pantalla el valor de cada nodo quisiéramos sumarlos? La respuesta es simple: cambiar la implementación, esto es, programar una nueva implementación de la *interface* `Proceso`, como veremos a continuación.

```
package libro.cap18.arbolbinario;

public class ProcesoImpleSuma implements Proceso<Integer>
{
   // variable de instancia para acumular los valores de los nodos
   private int suma = 0;

   public void ejecutar(Nodo<Integer> n)
   {
      // sumamos el valor del nodo n
      suma+=n.getV();
   }

   public int getSuma()
   {
      return suma;
   }
}
```

Ahora veamos el programa que recorre los nodos del árbol y suma sus valores:

```
   public static void main(String[] args)
   {
      // instanciamos la implementacion de ProcesaNodo
      ProcesoImpleSuma impleSuma = new ProcesoImpleSuma();

      // creamos el arbol
      Nodo<Integer> raiz = crearArbol();

      // lo recorremos y lo procesamos por niveles
      UArbol.amplitud(raiz, impleSuma);

      // mostramos el resultado luego del proceso
      System.out.println("La suma es: "+impleSuma.getSuma());
   }
```

18.3.3 Recorridos preorden, postorden e inorden

Agreguemos a la clase `UArbol` tres métodos que permitan recorrer un árbol según estos criterios.

```
package libro.cap18.arbolbinario;

import java.util.LinkedList;
import java.util.Queue;

public class UArbol
{
   public static <T> void preOrden(Nodo<T> root, Proceso<T> p)
   {
      if( root!=null )
      {
         p.ejecutar(root);
         preOrden(root.getIzq(), p);
         preOrden(root.getDer(), p);
      }
   }

   public static <T> void postOrden(Nodo<T> root, Proceso<T> p)
   {
      if( root!=null )
      {
         postOrden(root.getIzq(), p);
         postOrden(root.getDer(), p);
         p.ejecutar(root);
      }
   }

   public static <T> void inOrden(Nodo<T> root, Proceso<T> p)
   {
      if( root!=null )
      {
         inOrden(root.getIzq(), p);
         p.ejecutar(root);
         inOrden(root.getDer(), p);
      }
   }

   //
   // :
   //
}
```

Como vemos, la implementación de estos métodos es exactamente la misma que analizamos más arriba cuando vimos los diagramas de sus correspondientes algoritmos.

18.3.4 Recorrido iterativo

Agreguemos ahora un método que implemente un recorrido *preorden* iterativo a través de los nodos del árbol binario. Analicemos su código y lo discutiremos luego.

```
package libro.cap18.arbolbinario;

import java.util.LinkedList;
import java.util.Queue;

public class UArbol
{
   public static <T> void iterativo(Nodo<T> root, Proceso<T> p)
   {
      Stack<Nodo<T>> pila = new Stack<Nodo<T>>();
      pila.push(root);

      while( !pila.isEmpty() )
      {
         Nodo<T> n = pila.pop();
         p.ejecutar(n);

         if( n.getDer()!=null )
         {
            pila.push(n.getDer());
         }

         if( n.getIzq()!=null )
         {
            pila.push(n.getIzq());
         }
      }
   }

   //
   // :
   //
}
```

Desde el punto de vista del usuario, el hecho de que la implementación del recorrido en *preorden* sea recursiva o iterativa es, en principio, anecdótico ya que los nodos se procesarán exactamente en el mismo orden.

Sin embargo, una implementación iterativa nos da la posibilidad de implementar un *iterator* que le permitirá al usuario "navegar" a través de los nodos del árbol. Algo similar al método `next` de la clase `UTree` que utilizamos en el Capítulo 16 para desarrollar el compresor basado en el algoritmo de Huffman.

18.3.5 Iteradores

Llamamos "iterador" a un objeto que permite recorrer secuencialmente los datos de una colección. El recorrido debe ser *forward only*, es decir, desde el primero hacia el último y una sola vez.

En Java esta característica está definida en la *interface* `Iterator` del paquete `java.util`, por lo que "implementar un iterador" significa desarrollar una clase que implemente dicha *interface*.

La *interface* provee tres métodos: `hasNext`, `next` y `remove`. A este último no le daremos importancia; solo nos concentraremos en los primeros dos.

El método `hasNext` debe retornar `true` o `false` según exista o no un próximo elemento en la colección. El método `next` retorna el próximo elemento.

Veamos una implementación de `Iterator` que realiza una recorrida iterativa por los nodos de un árbol binario. Observemos que en ningún lugar recibimos como parámetro la instancia de `Proceso` que veníamos utilizando para desentendernos del proceso que se debe aplicar a cada nodo. ¿Será que aquí no la necesitamos? ¿Por qué?

```
package libro.cap18.arbolbinario;

import java.util.Iterator;
import java.util.Stack;

public class IteratorArbolBinarioImple<T> implements Iterator<Nodo<T>>
{
   private Stack<Nodo<T>> pila = new Stack<Nodo<T>>();

   public IteratorArbolBinarioImple(Nodo<T> raiz)
   {
      pila.push(raiz);
   }

   public boolean hasNext()
   {
      return !pila.isEmpty();
   }

   public Nodo<T> next()
   {
         Nodo<T> n = pila.pop();

         if( n.getDer()!=null )
         {
            pila.push(n.getDer());
         }

         if( n.getIzq()!=null )
         {
            pila.push(n.getIzq());
         }

         return n;
   }

   // este metodo no lo implementaremos, o queda vacio y
   // disparamos una excepcion por si al usuario se le ocurre
   // invocarlo :O)
   public void remove()
   {
      throw new RuntimeException("Este metodo no esta implementado");
   }
}
```
■

Con esta implementación de `Iterator`, tal vez, deberíamos eliminar el método `iterativo` de la clase `UArbol` y, en su lugar, agregar el método `crearIterador`

que instancie y retorne un objeto de la clase `IteratorArbolBinarioImple`, es decir, un *factory method* a través del cual el usuario podrá instanciar un iterador sin tener que preocuparse por su implementación.

```
package libro.cap18.arbolbinario;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class UArbol
{
   public static <T> Iterator<Nodo<T>> crearIterador(Nodo<T> root)
   {
      return new IteratorArbolBinarioImple<T>(root);
   }

   //
   // :
   //
}
```

Más arriba habíamos preguntado las razones por las cuales no era necesario recibir una instancia de `Proceso` en la implementación de `Iterator`. La respuesta a esta pregunta la encontraremos en el siguiente programa que prueba el método `crearIterador`.

```
   public static void main(String[] args)
   {
      // creamos el arbol
      Nodo<Integer> raiz = crearArbol();

      // obtenemos un iterador que nos permitira recorrer el arbol
      Iterator<Nodo<Integer>> it = UArbol.crearIterator(raiz);

      // mientras haya elementos los procesamos
      while( it.hasNext() )
      {
         // obtenemos el siguiente nodo
         Nodo<Integer> n = it.next();

         // el proceso lo aplicamos directamente
         System.out.println(n.getV());
      }
   }
```

Como vemos, la posibilidad de acceder secuencialmente a los nodos del árbol nos permite ocuparnos "personalmente" del proceso. En otras palabras: dentro del `while` disponemos del nodo para hacer con él lo que tengamos que hacer.

18.3.6 Iteradores vs. callback methods

La técnica que usamos al definir la *interface* `Proceso` con el método `ejecutar` y luego invocarlo para desentendernos del proceso se llama "retrollamada" o *callback*.

Desde el punto de vista del usuario, él simplemente escribe una clase que implemente la *interface* `Proceso`, sobrescribe el método `ejecutar` y luego "mágicamente" el método se llama solo, una vez por cada uno de los nodos del árbol. El método `procesar` es un *callback method*.

La implementación del iterador es más cercana a la programación estructurada ya que le permite al programador procesar los nodos dentro de un ciclo de repeticiones tal como lo hacíamos al inicio del libro.

Una técnica no es mejor que la otra. Son diferentes y, según sea el caso, alguna podría resultar más apropiada.

Por ejemplo, la clase `UTree` con su método `next` me permitió proveerle al lector una manera fácil de recorrer las hojas del árbol Huffman. Probablemente una solución basada en *callback methods*, a esa altura del libro, solo le hubiera aportado confusión.

18.3.7 Enfoque basado en objetos

Hasta aquí utilizamos la clase `UArbol` para agrupar un conjunto de métodos estáticos, dejando en manos del usuario (programador que la utiliza) la responsabilidad de crear, armar y mantener la estructura de datos.

Los ejemplos del tema que veremos a continuación serán desarrollados con un enfoque más "orientado a objetos", encapsulando dentro de una clase la estructura de datos y la lógica con la que esta debe ser manipulada.

18.4 Árbol Binario de Búsqueda

Un Árbol Binario de Búsqueda (ABB) es un árbol binario en el que se verifica que el valor de cada "nodo padre" es mayor que el valor de su "hijo izquierdo" pero es menor o igual que el valor de su "hijo derecho". Cuando un árbol cumple con esta propiedad resulta que al recorrerlo *inorden* sus nodos aparecerán ordenados.

Además, el hecho de que el valor del padre se ubique exactamente en el medio respecto de los valores de sus hijos hace que el conjunto se divida en "mitades". Es decir: cualquiera sea el valor de un nodo, todos los nodos ubicados a su izquierda tendrán valores menores y todos los valores de los nodos que se encuentren a su derecha serán mayores o iguales que este. Esta característica nos permitirá realizar búsquedas con una eficiencia comparable a la de la "búsqueda binaria" que estudiamos en el capítulo de operaciones sobre *arrays*.

En resumen, el árbol binario de búsqueda permite ordenar un conjunto de valores y permite también determinar, con extrema eficiencia, si el conjunto contiene o no a un determinado valor.

18.4.1 Crear un Árbol Binario de Búsqueda (ABB)

Crearemos un ABB a partir de los valores del siguiente conjunto:

`A` = {5, 2, 8, 1, 9, 7, 3, 4, 10, 6}

Los nodos deben agregarse respetando el orden original del conjunto y, como veremos a continuación, cualquiera sea el nodo "padre" su valor debe ser mayor que el valor de su hijo izquierdo pero menor o igual que el valor de su hijo derecho.

El árbol resultante según los elementos del conjunto `A` será el siguiente:

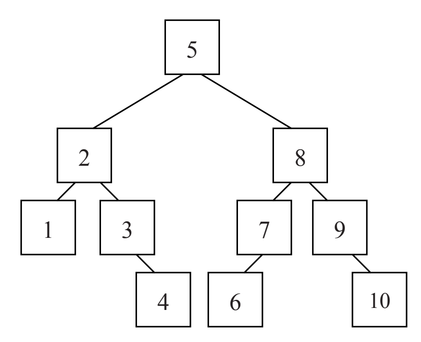


Fig. 18.10 Árbol binario de búsqueda con los elementos del conjunto A.

Hagamos un seguimiento “paso a paso” para analizar cómo se crea este ABB a partir de los valores del conjunto `A`.

Recordemos los valores del conjunto:

`A` = {5, 2, 8, 1, 9, 7, 3, 4, 10, 6}

Comenzamos creando un nodo con valor 5 que, por ser el primer valor del conjunto, se convertirá en la raíz del árbol binario.



Fig. 18.11 ABB con un único elemento.

El siguiente elemento del conjunto es 2 que, al ser menor que 5, lo agregaremos como su hijo izquierdo.

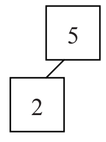


Fig. 18.12 ABB con dos elementos.

El siguiente valor que ingresará será 8 que, al ser mayor que 5, lo colocaremos como su hijo derecho.

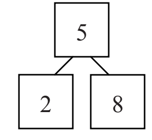


Fig. 18.13 ABB con tres elementos.

Ahora tenemos que agregar el 1. Como 1 es menor que 5, avanzamos sobre su hijo izquierdo. Luego, como 1 es menor que 2, lo colocamos como su hijo izquierdo.

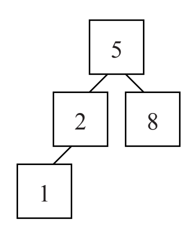


Fig. 18.14 ABB con cuatro elementos.

A continuación, ingresarán los valores 9 y 7. Como 9 es mayor que 5 y también es mayor que 8, lo colocaremos como hijo derecho de este. En cambio 7, que es mayor que 5, resulta ser menor que 8. Entonces lo ubicaremos como su hijo izquierdo.

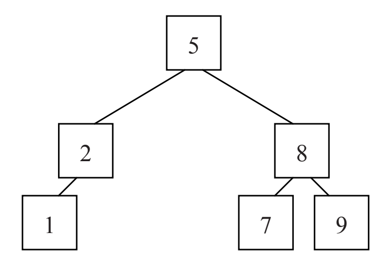


Fig. 18.15 ABB con seis elementos.

Ahora tenemos que agregar el valor 3. Resulta que 3 es menor que 5 y es mayor que 2. Será el hijo derecho de 2.

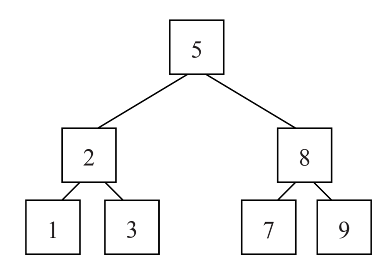


Fig. 18.16 ABB con siete elementos.

Solo resta agregar los valores 4, 10 y 6. Comencemos con 4: es menor que 5, mayor que 2 y mayor que 3: será hijo derecho de 3.

El 10 es mayor que 5 y también es mayor que 8 y 9: será hijo derecho de 9.

Por último, 6 es mayor que 5 pero es menor que 8 y 7: será hijo izquierdo de 7.

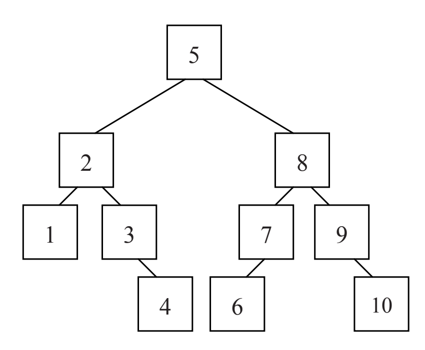


Fig. 18.17 ABB completo.

18.4.2 Encapsulamiento de la lógica y la estructura de datos (clase `Abb`)

Crearemos la clase `Abb` para encapsular la lógica y la estructura de un árbol binario de búsqueda.

La clase tendrá dos variables de instancia:

- La raíz del árbol (`root`) que inicialmente será `null`, y
- una instancia de `Comparator` (`comp`) que deberá proveer el usuario a través del constructor y que utilizaremos para comparar los valores de los nodos.

Respecto de los métodos, básicamente serán tres:

- `agregar`, que agregará un valor al ABB,
- `buscar`, que retornará un puntero al nodo que contenga un determinado valor o `null` si este valor no coincide con el valor de ninguno de los nodos del árbol y
- `eliminar`, que eliminará el nodo que contenga un valor especificado.

Opcionalmente el lector podrá agregar un cuarto método que, mediante un iterador o mediante la técnica de retrollamadas, implemente un recorrido *inorden* sobre los nodos del ABB para darle al usuario la posibilidad de acceder en orden a los elementos del conjunto contenido en el árbol. En este apartado se pueden dejar planteados los siguientes interrogantes: ¿Cuál será el criterio de ordenamiento de los nodos del árbol? ¿Qué lo determinará?

Comencemos con la estructura de la clase `Abb` que será genérica en `T` y, como dijimos más arriba, tendrá las variables de instancia: `root`, la raíz del árbol, y `comp`, una instancia de `Comparator` que el usuario debe proveer a través del constructor.

```
package libro.cap18.arbolbinario.abb;

import java.util.Comparator;
import libro.cap18.arbolbinario.Nodo;

public class Abb<T>
{
   // raiz del arbol binario
   private Nodo<T> root = null;

   // comparador que permitira comparar los elementos del arbol
   private Comparator<T> comp = null;
```

```
// constructor
public Abb(Comparator<T> c)
{
   this.comp = c;
}

// getter de la raiz, no haremos el setter
public Nodo<T> getRoot()
{
   return root;
}

// sigue mas abajo...
// :
```

De más está decir que el hecho de que la clase `Abb` sea genérica nos permitirá utilizar el árbol para trabajar con datos de cualquier tipo (enteros, cadenas, objetos, etc.).

18.4.3 Agregar un elemento al ABB (método `agregar`)

Ahora analizaremos el método `agregar` que recibirá un valor `v` de tipo `T` y lo agregará al árbol según el siguiente algoritmo:

- Crear un nuevo nodo con el valor `v` que se desea agregar.
- Comenzando desde la raíz (que la llamaremos `aux`) y mientras tenga hijos:
  - Si `v` es menor que el valor de `aux` entonces avanzar sobre su hijo izquierdo.
  - Si `v` es mayor o igual que el valor de `aux` entonces avanzar sobre su hijo derecho.
  - Así hasta llegar a un nodo que tenga libre su hijo izquierdo o derecho, según corresponda en función del valor `v`.
- Luego enlazar al nuevo nodo.

Veamos:

```
// :
// viene de mas arriba

public void agregar(T v)
{
   // nuevo hijo
   Nodo<T> nuevoHijo = new Nodo<T>(v);

   // si es el primero entonces sera la raiz
   if( root==null )
   {
      root = nuevoHijo;
      return;
   }

   // puntero auxiliar para recorrer el arbol desde la raiz
   Nodo<T> aux = root;
```

```
      // puntero a cada vertice o padre
      Nodo<T> padre = null;

      while( aux!=null )
      {
         // para este padre...
         padre = aux;

         // si el valor del nuevo hijo es menor...
         if( comp.compare(v, padre.getV())<0 )
         {
            // avanzo por la izquierda
            aux =aux.getIzq();
         }
         else
         {
            aux = aux.getDer();
         }
      }

      // si v < padre.getV() => lo agregamos a la izquierda
      if( comp.compare(v, padre.getV())<0 )
      {
         padre.setIzq(new Nodo<T>(v));
      }
      else
      {
         padre.setDer(new Nodo<T>(v));
      }
   }

   // sigue mas abajo
   // :
```

Observemos que al tratarse de una clase genérica resulta inevitable realizar las comparaciones a través del compadador `comp`.

18.4.4 Ordenar valores mediante un ABB (recorrido `inOrden`)

Como comentamos más arriba, el recorrido *inorden* sobre un ABB ordena los elementos del conjunto. Para probarlo vamos a crear un ABB con los valores del conjunto `A` y luego los mostraremos ordenados por consola .

Utilizaremos el método `inOrden` de la clase utilitaria `UArbol` y una instancia de la clase `ProcesoImpleImprime`, todos recursos que analizamos y utilizamos al inicio de este capítulo.

```
package libro.cap18.arbolbinario.abb;

import libro.cap18.arbolbinario.UArbol;
import libro.cap18.arbolbinario.ProcesoImpleImprime;
```

```
public class TestABB
{
   public static void main(String[] args)
   {
      // instanciamos el ABB con una instancia de Comparator
      // el codigo de CompInteger esta mas abajo...
      Abb<Integer> abb = new Abb<Integer>(new CompInteger());

      // recordemos el contenido del conjunto A
      // A = {5, 2, 8, 1, 9, 7, 3, 4, 10, 6}

      abb.add(5);
      abb.add(2);
      abb.add(8);
      abb.add(1);
      abb.add(9);
      abb.add(7);
      abb.add(3);
      abb.add(4);
      abb.add(10);
      abb.add(6);

      // recorremos inOrden y mostramos por pantalla el valor de cada nodo
      UArbol.inOrden(abb.getRoot(), new ProcesoImpleImprime<Integer>());
   }
}
```

El constructor de `Abb` espera recibir una instancia de `Comparator` que le permita al árbol comparar dos valores. Como los elementos del conjunto son de tipo `int` desarrollaremos la clase `CompInteger` de la siguiente manera:

```
package libro.cap18.arbolbinario.abb;

import java.util.Comparator;

public class CompInteger implements Comparator<Integer>
{
   public int compare(Integer i1, Integer i2)
   {
      return i1.compareTo(i2);
   }
}
```

Ahora sí, la salida del programa será:

```
1
2
3
:
10
```

18.4.5 Búsqueda de un elemento sobre un ABB (método `buscar`)

Cualquier nodo de un ABB tiene hijos con valores menores a su izquierda e hijos con valores mayores o iguales a su derecha. Esta característica permite garantizar que, dado un árbol binario de búsqueda `a`, un valor `v` y cualquiera de sus nodos `n`, entonces:

si `v<n.getV()` será porque `a` contiene a `v` en `n.getIzq()` o en alguno de sus hijos, descartando así a todo el subárbol que se desprende de `n.getDer()`.

Supongamos que, efectivamente, se verifica que `v<n.getV()`, entonces avanzaremos sobre su rama izquierda asignando `n = n.getIzq()`. Luego podemos volver a comparar y así continuaremos descartando una u otra mitad de árbol hasta encontrar lo que buscamos o hasta determinar que el árbol no contiene al elemento `v`.

Recordemos el ABB con el que venimos trabajando:

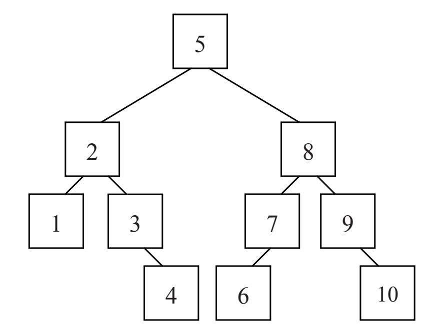


Fig. 18.18 Árbol binario de búsqueda.

Por ejemplo, para determinar si el árbol contiene el valor 9 entonces evaluamos el valor del nodo raíz, que es 5. Como 9 es mayor que 5 debería estar ubicado a su derecha. Luego avanzamos al nodo con valor 8, descartando definitivamente los nodos 2, 1, 3 y 4. Como 9 es mayor que 8 debería ubicarse a la derecha de este; por lo tanto, avanzamos sobre su hijo derecho, descartando así a los nodos 7 y 6. Luego, al comparar podremos determinar que el árbol contiene el valor 9. Solo necesitamos 3 comparaciones.

Busquemos el valor 6: es mayor que 5 (descartamos 2, 1, 3 y 4), es menor que 8 (descartamos 9 y 10), es menor que 7, es igual a 6. Necesitamos realizar 4 comparaciones.

Busquemos ahora un valor que no esté contenido en el árbol. Por ejemplo, 3.5. Este valor es menor que 5 (descartamos los nodos 8, 7, 9, 6 y 10), mayor que 2 (descartamos el nodo 1), mayor que 3 y menor que 4. Fueron necesarias 4 comparaciones para determinar que el árbol no contiene a dicho valor.

La búsqueda de un elemento en un ABB es altamente eficiente, comparable a la búsqueda binaria que estudiamos en el capítulo de operaciones sobre *arrays*.

Agregaremos el método `buscar` a la clase `Abb` que retornará un puntero al nodo que contiene el valor que buscamos o `null` si ninguno de los nodos del árbol contiene a dicho valor.

```
//:
// viene de mas arriba

public Nodo<T> buscar(T v)
{
   Nodo<T> aux = root;

   // mientras no sea null y mientras el valor de aux sea distinto de v
   while( aux!=null && comp.compare(v, aux.getV())!= 0 )
   {
```

```
            if( comp.compare(v, aux.getV())<0 )
            {
               aux = aux.getIzq();
            }
            else
            {
               if( comp.compare(v, aux.getV())>0 )
               {
                  aux = aux.getDer();
               }
            }
         }

         return aux;
      }
   }  // cierra la clase Abb
```

18.4.6 Eliminar un elemento del ABB (método `eliminar`)

Eliminar un nodo en un ABB no es tan simple como agregar o buscar un determinado valor. Básicamente la idea es permutar el nodo que vamos a eliminar por el nodo “más a la izquierda” de su subárbol derecho o bien permutarlo por el nodo “más a la derecha” de su subárbol izquierdo. Luego repetir este proceso tomando como raíz del árbol la nueva posición del nodo hasta que este se haya convertido en una hoja. Allí, simplemente, lo eliminamos.

Veamos algunos ejemplos:

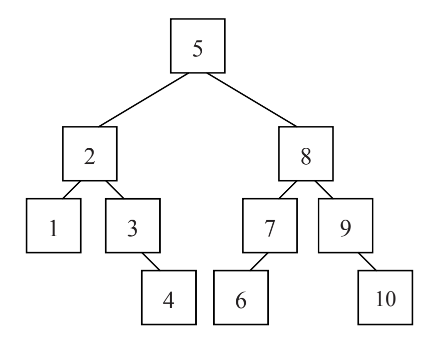


Fig. 18.19 Eliminamos el nodo 8.

Si queremos eliminar el nodo 8 del ABB que ilustra la figura tenemos que identificar el nodo “más a la izquierda” de su subárbol derecho (9) o el nodo “más a la derecha” de su subárbol izquierdo (7). Tomaremos esta opción; por lo tanto, permutamos el 8 por el 7 y repetimos el proceso.

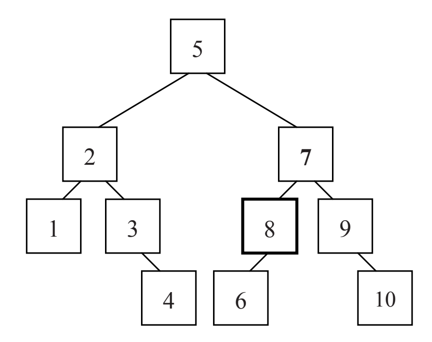


Fig. 18.20 Eliminamos el nodo 8.

Ahora, considerando como raíz al nodo 8, el nodo más a la derecha del subárbol izquierdo es 6. Entonces lo permutamos:

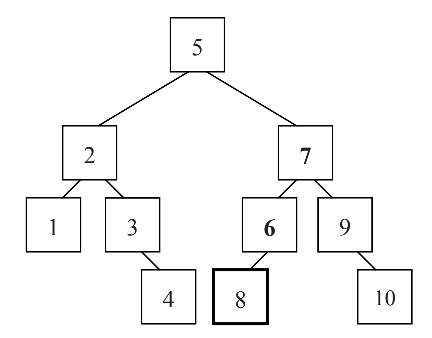


Fig. 18.21 – Eliminamos el nodo 8.

Ahora, simplemente podemos eliminarlo porque se convirtió en “hoja”. La implementación de este algoritmo quedará a cargo del lector.

18.5 Árbol n-ario

Un árbol *n-ario* es aquel en el que cada nodo puede tener diferentes cantidades de hijos. Veamos un ejemplo:

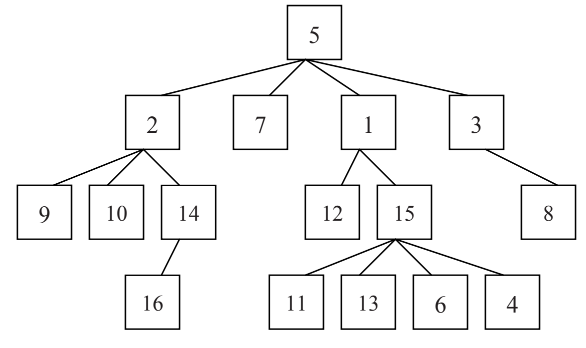


Fig. 18.22 Árbol *n-ario*.

Como vemos, en este caso el nodo raíz y el nodo 15 tienen 4 hijos. El nodo 2 tiene 3 hijos, el nodo 1 tiene 2, los nodos 3 y 14 tienen 1 y los nodos 7, 9, 10, 12, 8, 16, 11, 13, 6 y 4 no tienen hijos.

18.5.1 Nodo del árbol n-ario

El nodo de un árbol *n-ario* debe estar preparado para mantener una cantidad variable de hijos. Por esta razón las referencias generalmente se almacenan en un *array*.

Veamos algunos ejemplos:

Implementación en C

```
typedef struct NodoN
{
  int v;
  struct NodoN* hijos[];
}NodoN;
```

Implementación en Java

```
public class NodoN
{
  private int v;
  private NodoN[] hijos;

  // setters y getters
}
```

En Java usando *generics*

```
public class NodoN<T>
{
  private T v;
  private NodoN<T>[] hijos;

  // setters y getters
}
```

En Java usando *generics* y `ArrayList`

```
public class NodoN<T>
{
  private T v;
  private ArrayList<NodoN<T>> hijos;

  // setters y getters
}
```

18.5.2 Recorridos sobre un árbol n-ario

Al igual que con los árboles binarios, podemos emprender recorridos *pre* y *postorden* sobre los nodos del árbol *n-ario*.

Por ejemplo, un recorrido en *preorden* sobre los nodos del árbol de la figura anterior sería el siguiente: 5, 2, 9, 10, 14, 16, 7, 1, 12, 15, 11, 13, 6, 4, 3, 8.

Veamos los algoritmos:

```
public static <T> void preOrden(NodoN<T> root, Proceso<T> p)
{
   if( root!=null )
   {
      // procesamos el nodo
      p.ejecutar(root);

      // invocamos recursivo con cada uno de sus hijos
      for(NodoN<T> hijo: root.getHijos())
      {
         preOrden(hijo, p);
      }
}
```

```
public static <T> void postOrden(Nodo<T> root, Proceso<T> p)
{
   if( root!=null )
   {
      // invocamos recursivo con cada uno de los hijos
      for(NodoN<T> hijo: root.getHijos())
      {
         postOrden(hijo, p);
      }
      // procesamos el nodo
      p.ejecutar(root);
   }
}
```

■

Ahora recordemos el algoritmo con el que implementamos el recorrido *inorden* sobre los nodos de un árbol binario:

```
public static <T> void inOrden(Nodo<T> root, Proceso<T> p)
{
   if( root!=null )
   {
      inOrden(root.getIzq(), p);
      p.ejecutar(root);
      inOrden(root.getDer(), p);
   }
}
```

■

¿Podríamos implementar un recorrido *inorden* sobre los nodos de un árbol *n-ario*? Dejo planteado este interrogante para el lector.

18.5.3 Permutar los caracteres de una cadena

Utilizando un árbol *n-ario* también podemos resolver el problema de encontrar todas las permutaciones que existen entre los caracteres de una cadena.

Recordemos el enunciado: desarrollar un programa que muestre por consola todas las permutaciones que se pueden realizar con los caracteres de una cadena de cualquier longitud. La cadena la ingresará el usuario por consola o por línea de comandos y no tendrá caracteres repetidos.

Análisis

Supongamos que la cadena que vamos a procesar es “ABCD”. Podemos simplificar el problema pensando que varias de las permutaciones comenzarán con ‘A’, es decir, el carácter ‘A’ seguido de todas las permutaciones posibles entre “BCD”. Incluso, podemos simplificarlo más aun si pensamos que muchas de estas combinaciones serán precedidas por el carácter ‘B’, es decir, ‘B’ más todas las permutaciones que se puedan realizar entre los caracteres de “CD”.

Es decir, cada carácter precederá las permutaciones de sus hermanos. Una estructura de árbol *n-ario* nos ayudará a representar la idea.

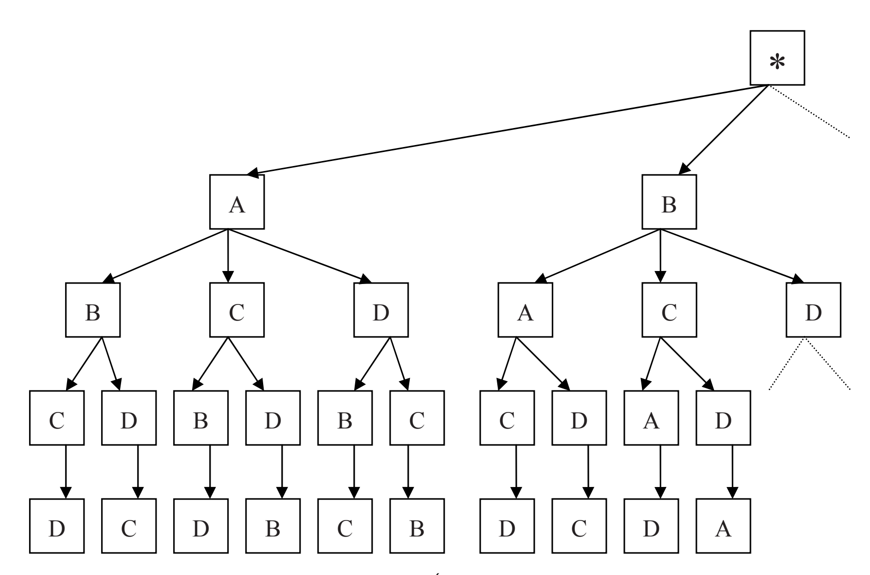


Fig. 18.23 Árbol *n-ario*.

En la figura vemos una parte del árbol que separa los caracteres de la cadena. Luego, recorriendo el árbol desde arriba hacia abajo y de izquierda a derecha, podemos obtener todas las permutaciones posibles.

Por ejemplo: ABCD, ABDC, ACBD, ACDB, ADBC, ADCB, BACD, BADC, BCAD, BCDA...

El desarrollo del ejercicio quedará a cargo del lector.

18.5.4 Implementación de un “AutoSuggest”

Hace unos años Google nos sorprendió al hacer que la lista de resultados de una búsqueda se desplegara mientras íbamos escribiendo cada letra en el buscador. Al teclear cada carácter la lista se refinaba un poco más haciendo que el conjunto de resultados fuera más acotado y preciso. Pronto esta característica se denominó *AutoSuggest* y se expandió por todos los sitios de Internet.

Veamos cómo podemos implementar este mecanismo para filtrar, por ejemplo, un conjunto de nombres contenidos en un *array*.

Supongamos que el *array* `arr` tiene la siguiente lista de nombres:

```
arr = { "Pablo" , "Paula" , "Alberto"
      , "Analia", "Belen" , "Alejandra"
      , "Paola" , "Andres", "Pedro" }
```

La estrategia consiste en armar un árbol *n-ario* que nos permita indexar los nombres de la lista.

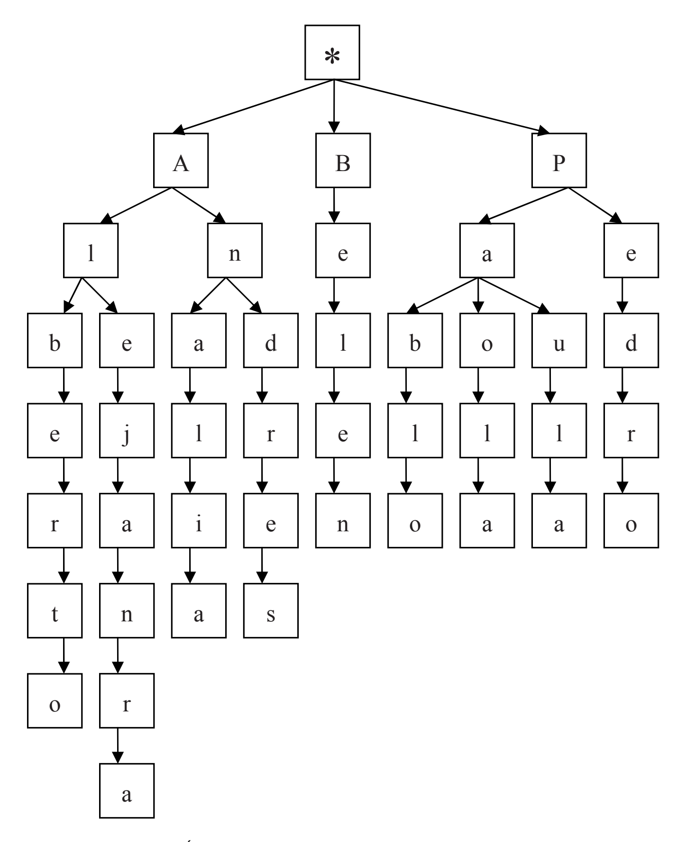


Fig. 18.24 Árbol *n-ario* que indexa la lista de sugerencias.

Los diferentes caminos que podemos tomar yendo desde la raíz hacia las hojas y de izquierda a derecha forman el conjunto de sugerencias que concuerdan con cada prefijo.

Por ejemplo, si el prefijo es “P” entonces desde la raíz del árbol nos movemos hasta el hijo identificado con ese carácter. Luego, todos los caminos que podemos tomar yendo desde este nodo hacia abajo y de izquierda a derecha nos llevarán a formar todas las palabras que comienzan con “P”.

Ahora, si el prefijo fuese “Pa” entonces el primer carácter nos hará desplazar hasta el hijo identificado por “P”. Habiéndonos posicionado en este nodo, el segundo carácter nos desplazará hasta su hijo identificado por “a”. Luego, todos los caminos que surgen a partir de este nodo, de arriba hacia abajo y de izquierda a derecha, nos permitirán formar las sugerencias que comienzan con “Pa”.

Para finalizar, si el prefijo fuese vacío entonces simplemente nos quedamos en la raíz del árbol y, desde allí, tenemos acceso a la lista completa de nombres.

Notemos que cada nodo se identifica con un carácter y tiene una lista de hijos. Una forma simple de implementarlo será utilizando una *hashtable*.

```
package libro.cap18.autosuggest;
public class Nodo extends Hashtable<Character, Nodo>
{
}
```

■

Dejaré planteada la estructura de la clase `AutoSuggest` pero el desarrollo de cada uno de los métodos quedará a cargo del lector.

```
package libro.cap18.autosuggest;

public class AutoSuggest
{
   private Nodo root = null;

   // recibe un array de cadenas y debe crear el arbol cuya raiz sera root
   public void indexar(String[] s)
   {
      // programar aqui...
   }

   // recibe un prefijo y debe retornar un String[] que contenga
   // el conjunto de palabras indexadas que comienzan con el prefijo
   // especificado
   public String[] buscar(String prefix)
   {
      // programar aqui...
   }

   // prueba el funcionamiento de la clase
   public static void main(String[] args)
   {
      AutoSuggest a = new AutoSuggest();
      arr = { "Pablo" , "Paula" , "Alberto"
            , "Analia", "Belen" , "Alejandra"
            , "Paola" , "Andres", "Pedro" };

      // indexamos las palabras del array
      a.indexar(arr);

      // obtenemos aquellas que comienzan con un determinado prefijo
      String res[] = a.mostrar("An");

      // las mostramos
      for(String s:res)
      {
         System.out.println(s);
      }
   }
}
```

18.6 Resumen

Los árboles constituyen la estructura de datos fundamental de muchos algoritmos. Si bien su naturaleza recursiva nos induce a tratarlos con algoritmos recursivos, hemos visto también que podemos encontrar métodos iterativos equivalentes. Eso sí, todos estos montados sobre una estructura de pila.

Es decir, la pila siempre estará presente cuando trabajemos con árboles: podemos implementarla manualmente o bien indirectamente apilando invocaciones recursivas en el *stack* de llamadas, tal como lo estudiamos en el capítulo de recursividad.

El próximo paso será estudiar los diferentes métodos de ordenamiento. Entre ellos veremos un algoritmo llamado *heapsort* que utiliza un árbol binario como base para realizar sus operaciones. Este árbol debe cumplir con la condición de ser "montículo" (*heap* en inglés), tema que en este capítulo hemos pasado por alto pero que estudiaremos en detalle cuando llegue el momento de aplicarlo.

Como paso previo, estudiaremos un mecanismo que nos permitirá "medir" la eficiencia de un algoritmo. Luego, utilizaremos esta herramienta de medición llamada "complejidad algorítmica" para comparar los diferentes algoritmos de ordenamiento y entender por qué algunos llegan a ser extremadamente eficientes mientras que otros solo sirven para ordenar conjuntos muy pequeños.

18.7 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

18.7.1 Mapa conceptual

18.7.2 Autoevaluaciones

18.7.3 Presentaciones*

19

Complejidad algorítmica

Contenido

Objetivos del capítulo

- Entender al estudio de complejidad algorítmica como una herramienta de decisión orientada a predecir qué comportamiento tendrán nuestros algoritmos, al enfrentarse a situaciones adversas.
- Comparar algoritmos equivalentes para sacar conclusiones en base a su complejidad.

Competencias específicas

- Comprender la complejidad de los algoritmos e identificar la eficiencia de los mismos.

19.1 Introducción

En el primer capítulo de este libro, explicamos que un algoritmo es un conjunto finito de acciones que dan solución a un determinado problema y que si dos algoritmos diferentes resuelven el mismo problema entonces los llamamos algoritmos equivalentes.

La complejidad algorítmica permite establecer una comparación entre algoritmos equivalentes para determinar, en forma teórica, cuál tendrá mejor rendimiento en condiciones extremas y adversas.

Para esto, se trata de calcular cuantas instrucciones ejecutará el algoritmo en función del tamaño de los datos de entrada. Llamamos "instrucción" a la acción de asignar valor a una variable y a la realización de las operaciones aritméticas y lógicas.

Como resultaría imposible medir el tiempo que demora una computadora en ejecutar una instrucción, simplemente diremos que cada una demora 1 unidad de tiempo en ejecutarse. Luego, el algoritmo más eficiente será aquel que requiera menor cantidad de unidades de tiempo para concretar su tarea.

19.2 Conceptos iniciales

Hecha esta introducción, analicemos el siguiente algoritmo que implementa la búsqueda secuencial de un elemento dentro de un *array* de longitud len .

19.2.1 Análisis del algoritmo de la búsqueda secuencial

Recordemos el algoritmo.

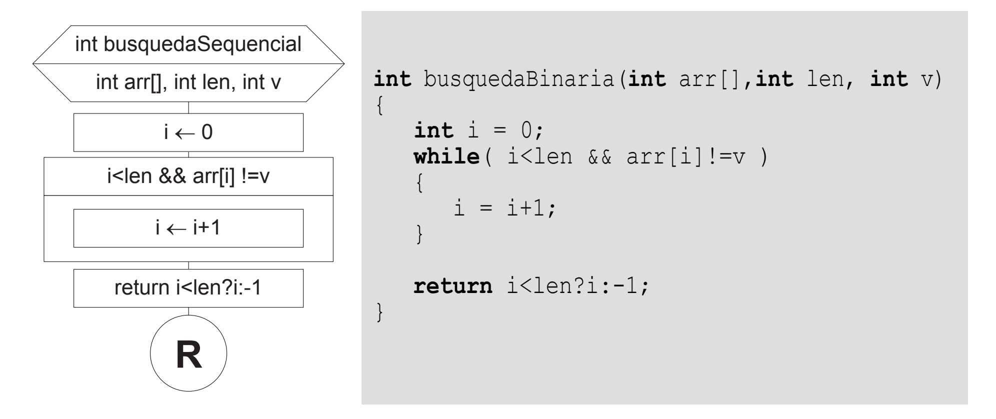


Fig. 19.1 Búsqueda secuencial de un elemento en un *array*.

El algoritmo comienza asignando el valor 0 a la variable `i` (1 unidad de tiempo). Luego, para evaluar la cabecera del ciclo iterativo debe realizar las siguientes instrucciones, cada una de las cuales requiere 1 unidad de tiempo:

`i<len`	Operación lógica
`arr[i]`	Indexación del *array*, operación aritmética que equivale a: `arr+i` (aritmética de direcciones, Capítulo 6)
`arr[i]!=v`	Operación lógica
`i<len && arr[i]!=v`	"and", operación lógica

Dentro del ciclo `while`, si corresponde, se ejecutan dos instrucciones: `i+1` y la asignación del resultado de esta operación a la variable `i` (en total: 2 unidades de tiempo).

Fuera del ciclo se realiza un *if-inline* donde se evalúa la siguiente expresión lógica: `i<len` (1 unidad de tiempo) y luego el `return` (1 unidad de tiempo).

La asignación `i=0`, el *if-inline* y el `return` se harán siempre. La evaluación de la cabecera del ciclo `while` se ejecutará al menos 1 vez. Luego, en función de los valores de los parámetros se ingresará o no al cuerpo del `while` donde, por cada iteración, se incrementará el valor del a variable `i` y se volverá a evaluar la condición de la cabecera.

En resumen, las instrucciones que el algoritmo ejecutará sí o sí, independientemente de cuáles sean los datos de entrada, son las siguientes:

Instrucción	`i=0`	`i<len && arr[i]!=v`	`return i<len?i:-1`
Unidades de tiempo	1	4	2

Y las instrucciones que se ejecutarán dependiendo de los valores de los datos de entrada son:

Instrucción	`i = i+1`	`i<len && arr[i]!=v`
Unidades de tiempo	$2n$	$4n$

Donde n es la cantidad de iteraciones que realice el `while`.

Luego, si sumamos las instrucciones fijas más las instrucciones que dependen de la cantidad de iteraciones, hallaremos a la función $f(n)$ que expresa la complejidad del algoritmo: $f(n) = 6n+7$.

Una vez hallada la función podemos analizar diferentes escenarios. Por ejemplo, si el elemento que buscamos se encuentra en la primera posición del *array* entonces el algoritmo no ingresará al `while` y solo requerirá 7 unidades de tiempo. Claramente este es el mejor de los casos.

La peor situación se dará cuando el valor que buscamos no exista dentro del *array* o cuando se encuentre en la última posición. En este caso el algoritmo requerirá 7 unidades de tiempo fijas más $6n$ unidades de tiempo, siendo n igual a `len`.

19.3 Notación O grande (cota superior asintótica)

En general, todos los algoritmos suelen ser óptimos cuando las condiciones son favorables, pero cuando las condiciones son adversas los problemas de *performance* se ponen en evidencia.

Como decíamos más arriba, una unidad de tiempo por sí misma es imperceptible. Por esto, si un algoritmo requiere 1, 2, 3, o 100 unidades de tiempo más que otro no notaremos ninguna diferencia. Lo importante en este tipo de análisis es poder encontrar una función que acote el crecimiento que, en el peor de los casos, tendrá la complejidad del algoritmo.

Al conjunto de funciones que acotan dicho crecimiento lo llamamos $O(g(x))$ donde x representa la cantidad de elementos que se van a procesar y $g(x)$ es cualquier cota superior de la función de complejidad del algoritmo.

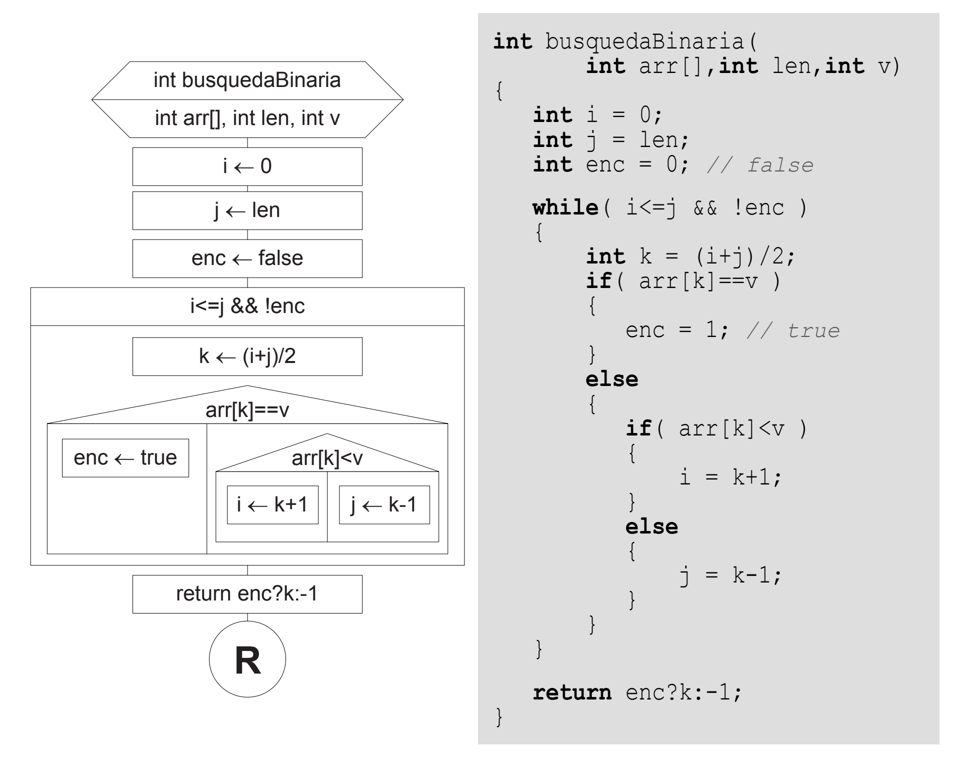
Más formalmente, lo plantearemos de la siguiente manera: sea $f(x)$ la función que expresa la complejidad de un algoritmo, entonces la expresión $f(x)$ pertenece a $O(g(x))$, significa que $f(x)$ crece, a lo sumo, tan rápido como cualquiera de las funciones g del conjunto O.

Volviendo al caso de la búsqueda secuencial de un elemento dentro de un *array* cuya función de complejidad es 6*n*+7, podemos simplificarlo y simplemente decir que el algoritmo es de orden lineal o que tiene una complejidad lineal.

Ahora vamos a analizar el algoritmo de la búsqueda binaria para hallar su orden de complejidad y luego contrastarlo con el de la búsqueda secuencial.

19.3.1 Análisis del algoritmo de la búsqueda binaria

Recordemos el algoritmo que estudiamos anteriormente.




```
int busquedaBinaria(
       int arr[],int len,int v)
{
   int i = 0;
   int j = len;
   int enc = 0; // false

   while( i<=j && !enc )
   {
       int k = (i+j)/2;
       if( arr[k]==v )
       {
          enc = 1; // true
       }
       else
       {
          if( arr[k]<v )
          {
              i = k+1;
          }
          else
          {
              j = k-1;
          }
       }
   }

   return enc?k:-1;
}
```

Fig. 19.2 Búsqueda secuencial de un elemento en un *array*.

Obviemos las instrucciones fijas y concentrémonos directamente en la cantidad de iteraciones que realizará el ciclo `while`.

En la primera iteración será `i=0` y `j=len`. Esto hará que el algoritmo procese los `len` elementos del *array*.

En la segunda iteración, si corresponde, se procesará la mitad del *array* ya que habremos desplazado alguno de los índices `i`, `j`. Procesaremos `len`/2 elementos.

En la tercera iteración, si corresponde, procesaremos la mitad de la mitad del *array*, es decir, `len`/2/2, lo que equivale a decir `len`/2^2 elementos.

En la cuarta iteración, siempre que corresponda, procesaremos la mitad de la mitad de la mitad del *array*, es decir, `len`/2/2/2 = `len`/2^3.

Generalizando, en la *n-ésima* iteración estaremos procesando `len`/2^n elementos.

El peor de los casos se dará cuando `len`/2^n sea igual a 1 ya que en esta situación aún no habremos encontrado el elemento que buscamos y no podremos seguir dividiendo el *array*.

Ahora bien,

si:	$1 = len/2^n$	entonces
pasamos multiplicando 2^n:	$2^n = len$	luego,
aplicando log_2 en ambos lados:	$log_2(2^n) = log_2(len)$	
resolvemos:	$n = log_2(len)$	

Recordemos que *n* es la cantidad de iteraciones que, en el peor de los casos, realiza el `while`; por lo tanto, la función de complejidad de la búsqueda binaria es $log_2(k)$, siendo *k*=`len`. Para simplificarlo diremos que el algoritmo tiene una complejidad logarítmica.

Comparemos las gráficas de las funciones de la búsqueda binaria y la búsqueda secuencial.

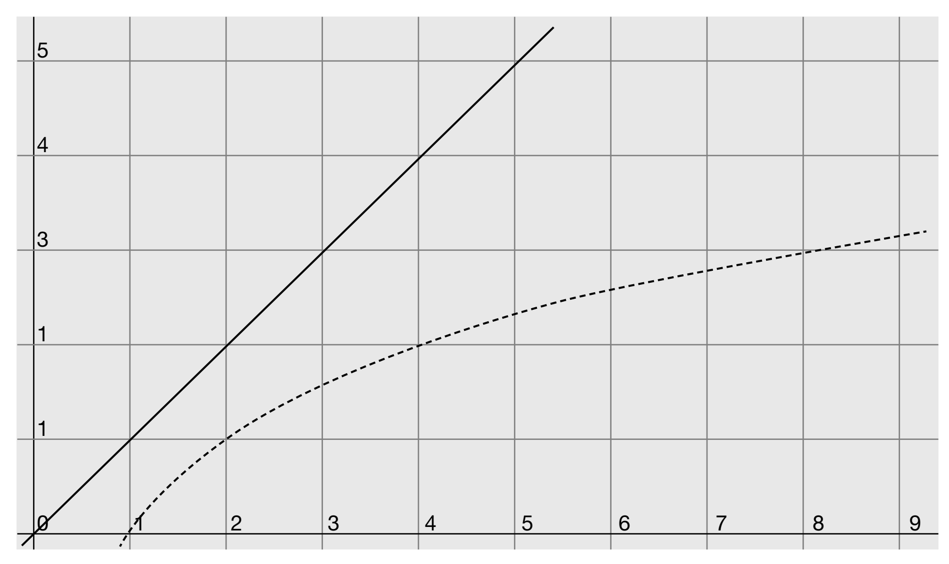
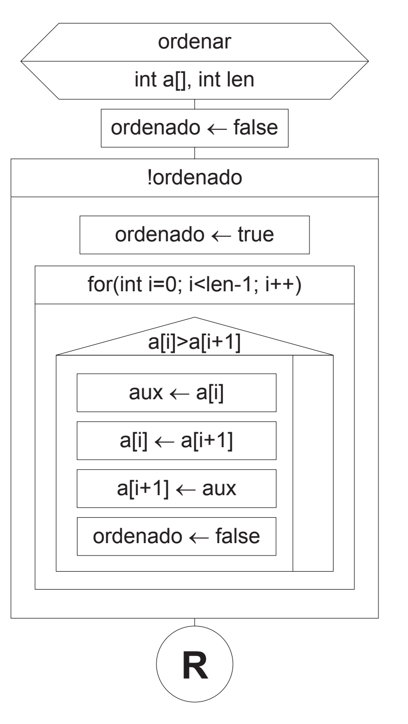


Fig. 19.3 Comparación de la complejidad de los algoritmos de búsqueda.

Como vemos, la función de la búsqueda binaria tiene un crecimiento mucho más atenuado que la función de la búsqueda secuencial. Esto permite determinar que a medida que crece el tamaño del *array* (eje de las equis) requerirá realizar una menor cantidad de instrucciones.

19.3.2 Análisis del algoritmo de ordenamiento por burbujeo

Recordemos el algoritmo.




```
void ordenar(int a[],int len)
{
   int ordenado=0;

   while( !ordenado )
   {
      ordenado=1;

      for(int i=0; i<len-1; i++)
      {
         if( a[i]>a[i+1] )
         {
            int aux=a[i];
            a[i]=a[i+1];
            a[i+1]=aux;
            ordenado=0;
         }
      }
   }
}
```

Fig. 19.4 Algoritmo de ordenamiento por burbujeo.

En este algoritmo el ciclo interno ejecuta `len`-1 iteraciones por cada una de las "pasadas" que realicemos sobre el *array*. Recordemos que una "pasada" implica recorrer el *array* comparando cada elemento respecto del elemento siguiente para determinar si ambos están en orden o no y entonces permutarlos.

Si durante una pasada se realiza al menos una permutación se forzará la realización de una nueva pasada. Así hasta que no sea necesario permutar ningún elemento, situación que nos permitirá determinar que el *array* quedó ordenado.

El peor de los casos se dará cuando los elementos del *array* se encuentren justamente en el orden inverso al cual los queremos ordenar. Por ejemplo, el siguiente *array* está ordenado decrecientemente pero está totalmente desordenado respecto del orden ascendente.

```
arr = {5, 4, 3, 2, 1}
```

Si este fuera el caso, luego de la primera pasada del algoritmo de ordenamiento por burbujeo el *array* quedará así:

```
arr = {4, 3, 2, 1, 5}
```

Luego de segunda pasada quedará así:

```
arr = {3, 2, 1, 4, 5}
```

Luego de la tercera pasada será:

```
arr = {2, 1, 3, 4, 5}
```

Y una nueva pasada lo dejará así:

```
arr = {1, 2, 3, 4, 5}
```

Aunque el *array* quedó ordenado, la permutación del elemento 2 por el elemento 1 nos obligará a realizar una pasada adicional.

De este análisis se desprende que para ordenar un *array* de 5 elementos totalmente desordenados debemos realizar 5 pasadas y por cada una de estas el ciclo interno iterará 4 veces.

Genéricamente hablando, para un *array* de longitud n, en el peor de los casos debemos atenernos a $n(n-1)$ iteraciones del `for` interno, lo que equivale a decir: n^2-n iteraciones.

Dado que esta función está acotada superiormente por n^2 decimos que el ordenamiento por burbujeo tiene una complejidad cuadrática.

19.4 Cota inferior (Ω) y cota ajustada asintótica (Θ)

Así como $O(g(x))$ representa el conjunto de las funciones que acotan superiormente a la función de complejidad de un determinado algoritmo, $\Omega(g(x))$ representa al conjunto de las funciones que la acotan inferiormente. Además, $\Theta(g(x))$ representa al conjunto de las funciones que crecen a la misma velocidad.

La siguiente tabla nos permitirá ordenar estos conceptos.

$O(g(x))$	Cota superior asintótica	Cualquiera sea la función $g(x)$, crecerá más de prisa o, a lo sumo, igual que $f(x)$.
$\Omega(g(x))$	Cota inferior asintótica	Cualquiera sea la función $g(x)$, crecerá más lento o, a lo sumo, tan rápido como $f(x)$.
$\Theta(g(x))$	Cota ajustada asintótica	Cualquiera sea la función $g(x)$, crecerá a la misma velocidad que $f(x)$.

19.5 Resumen

El análisis de la complejidad algorítmica nos permite predecir el comportamiento que tendrá un algoritmo cuando se enfrente a situaciones adversas. Esto lo convierte en una herramienta de decisión importantísima que nos ayudará a optar entre una u otra implementación de algoritmos equivalentes.

En el próximo capítulo aplicaremos estos conceptos para analizar y comparar diferentes algoritmos de ordenamiento.

19.6 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

19.6.1 Mapa conceptual

19.6.2 Autoevaluaciones

19.6.3 Presentaciones*

20

Algoritmos de ordenamiento

Contenido

Objetivos del capítulo

- Conocer los diferentes algoritmos de ordenamiento: *bubble sort*, *selection sort*, *insertion sort*, *quicksort*, *heapsort*, *shellsort*, *binsort* y *radixsort*.
- Comparar estos algoritmos según su complejidad algorítmica.
- Analizar empíricamente su eficiencia, midiendo los tiempos que requieren para ordenar *arrays* de gran tamaño.

Competencias específicas

- Aplicar el método de ordenamiento pertinente en la solución de un problema real.

20.1 Introducción

En este capítulo estudiaremos diferentes métodos de ordenamiento de *arrays*. Dado que algunos de estos algoritmos son verdaderamente complejos resultan, por sí mismos, excelentes ejercicios de programación.

Por esto, aquí solo explicaremos su lógica, analizaremos su complejidad algorítmica e incluso mediremos su rendimiento, pero en todos los casos la implementación quedará a cargo del lector y constituirá la práctica recomendada del presente capítulo.

Veremos que los algoritmos más simples, como el caso de *bubble sort*, tienen un orden de complejidad cuadrática $O(n^2)$ mientras que los más eficientes, y también más difíciles de implementar, tienen una complejidad cuasi lineal $O(n\ log(n))$.

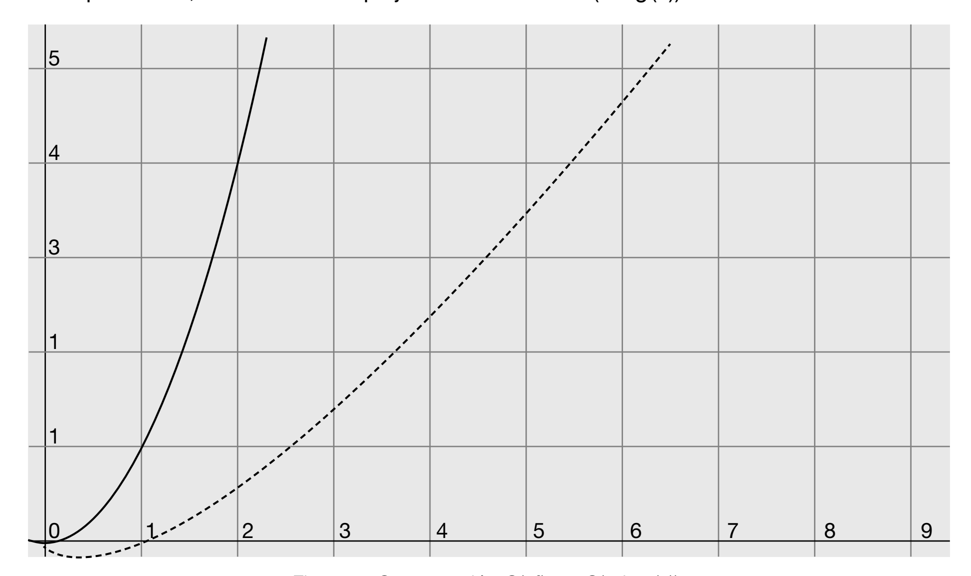


Fig. 20.1 Comparación $O(n^2)$ vs. $O(n\ log\ (n))$.

En este gráfico que contrasta la función cuadrática con la función cuasi lineal podemos apreciar que, a medida que aumenta la longitud del *array* (desplazándonos hacia la derecha), la función cuadrática devuelve valores mucho más elevados que la función cuasi lineal. El algoritmo cuadrático debe ejecutar una cantidad de instrucciones mucho mayor que el de orden cuasi lineal y esto hace que su rendimiento sea notablemente inferior.

Para facilitar la comprensión de los algoritmos de ordenamiento solo trabajaremos sobre *arrays* de valores enteros. El lector ya sabe que fácilmente, proveyendo una adecuada implementación de `Comparator`, se puede extender la funcionalidad de estos algoritmos a *arrays* de cualquier otro tipo de datos.

Antes de comenzar le sugerimos al lector que a medida que avance en la lectura de este capítulo vaya desarrollando la clase `Ordenamiento` completando los siguientes métodos estáticos.

```
public class Ordenamiento
{
   // implementacion de bubble sort
   public static void bubbleSort(int args[]){...}

   // implementacion de bubble sort optimizado
   public static void optimizedBubbleSort(int args[]){...}

   // implementacion del ordenamiento por seleccion
   public static void selectionSort(int args[]){...}

   // implementacion del ordenamiento por insercion
   public static void insertionSort(int args[]){...}

   // implementacion de quick sort
   public static void quickSort(int args[]){...}

   // implementacion del ordenamiento por monticulos
   public static void heapSort(int args[]){...}

   // implementacion de shell sort
   public static void shellSort(int args[]){...}

   // implementacion de bin sort
   public static void binSort(int args[]){...}

   // implementacion del ordenamiento de raiz
   public static void radixSort(int args[]){...}
}
```

20.2 Bubble sort (ordenamiento por burbujeo)

Consideremos el *array* `arr` de longitud `len`.

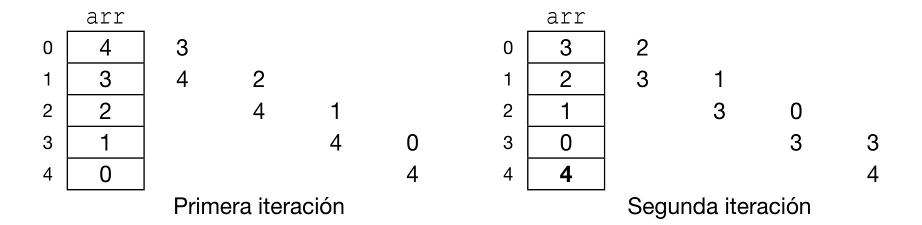
El algoritmo de la burbuja consiste en recorrer el *array* comparando el valor del *i-ésimo* elemento con el valor del elemento *i*+1 y, si estos se encuentran desordenados, entonces permutarlos.

Dicho de otro modo, comenzamos comparando `arr[0]` con `arr[1]`. Si `arr[0]>arr[1]` entonces los permutamos. Luego comparamos `arr[1]` con `arr[2]` y, si corresponde, también los permutamos y así hasta llegar a comparar `arr[len-2]` con `arr[len-1]`.

Luego de esta "pasada" o iteración a través de los elementos del *array*, el elemento de mayor valor o el más "pesado" quedará ubicado en la última posición.

El peor escenario con el que nos podemos encontrar es el de tener que ordenar un *array* completamente desordenado, es decir, en orden inverso.

Por ejemplo, si el *array* fuese: `arr = {4, 3, 2, 1, 0}`, entonces:



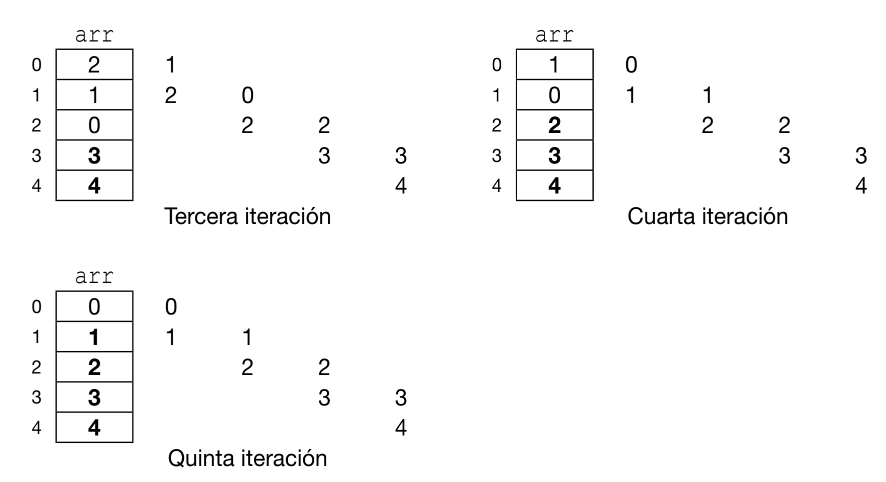


Fig. 20.2 Ordenamiento mediante *bubble sort*.

Como vemos, en cada iteración llevamos hacia abajo al elemento más pesado y solo podemos considerar que el *array* quedó ordenado luego de realizar una iteración en la que no haya sido necesario hacer ninguna permutación.

Así, para ordenar el *array* `arr` de longitud 5 necesitamos realizar 5 iteraciones, en cada una de las cuales haremos 4 comparaciones.

Si bien el algoritmo de ordenamiento por burbujeo es muy fácil de implementar, su complejidad cuadrática limita seriamente su capacidad para ordenar *arrays* de gran tamaño. Para probarlo empíricamente analizaremos su comportamiento al ordenar *arrays* de 10, 100, 1000, 10000, 100000 y 1000000 elementos.

Para generar *arrays* de semejantes tamaños desarrollaremos la clase utilitaria `UArray` con los siguientes métodos:

- `generarArray` – genera un *array* de cualquier longitud con valores aleatorios u ordenados de mayor a menor.
- `estaOrdenado` - retornará `true` o `false` según el *array* que reciba como parámetro esté o no ordenado ascendentemente.

```
public class UArray
{
   public static int[] generarArray(int n, boolean random)
   {
      int arr[]=new int[n];
      for( int i=0; i<n; i++ )
      {
         if( random )
         {
            arr[i]=(int)(Math.random()*n);
         }
         else
         {
            arr[i]=n-i;
         }
      }

      return arr;
   }
```

```
   public static boolean estaOrdenado(int arr[])
   {
      for( int i=0; i<arr.length-1; i++ )
      {
         if( arr[i+1]<arr[i] )
         {
            return false;
         }
      }

      return true;
   }
}
```

El método `estaOrdenado` será de mucha utilidad para poder verificar si las implementaciones de nuestros algoritmos funcionan correctamente con *arrays* de gran tamaño.

Utilizando la clase `UArray`, la clase `Performance` (que utilizamos en los capítulos anteriores) y una implementación básica de *bubble sort*, analicemos el siguiente programa que intenta ordenar *arrays* completamente desordenados de 10, 100, 1000, 10000, 100000 y 1000000 elementos.

```
public class TestBubble
{
   public static void main(String[] args)
   {
      Performance p=new Performance();
      int arr[];

      int cantElm[] = {10, 100, 1000, 10000, 100000, 1000000}

      for( int i=0; i<cantElm.length; i++ )
      {
         // generamos un array de n elementos (siendo n el valor de cantElm[i]
         arr = UArray.generarArray(cantElm[i], false);

         // lo ordenamos y medimos el tiempo empleado
         p.start();
         Ordenamiento.bubbleSort(arr);
         p.stop();

         // mostramos por consola el tiempo empleado
         System.out.print(p);

         // verificamos si el array quedo ordenado
         System.out.println(" "+UArray.estaOrdenado(arr));
      }
   }
}
```

Al ejecutar el programa en mi computadora Intel Core 2 Duo, 2.20 Ghz 2 GB RAM obtuve los siguientes resultados:

```
0 milisegundos (0 minutos, 0 segundos) true
0 milisegundos (0 minutos, 0 segundos) true
0 milisegundos (0 minutos, 0 segundos) true
329 milisegundos (0 minutos, 0 segundos) true
32031 milisegundos (0 minutos, 32 segundos) true
```

Las primeras tres líneas corresponden al ordenamiento de los *arrays* de 10, 100 y 1000 elementos. La cuarta línea corresponde al tiempo que *bubble sort* demandó para ordenar un *array* de 10000 elementos y la quinta línea indica el tiempo que el algoritmo requirió para ordenar un *array* de 100000 elementos.

Respecto del *array* de 1 millón de elementos, el proceso tiene ya más de 20 minutos ejecutándose en mi computadora y aún no termina, lo que demuestra que *bubble sort* funciona perfectamente para ordenar *arrays* de menos de 1000 elementos pero para *arrays* más grandes comienza a manifestar su ineficiencia..

20.2.1 Bubble sort optimizado

Si aprovechamos el hecho de que luego de cada iteración el elemento más "pesado" se ubica en la última posición del *array* (es decir, en su posición definitiva), podemos mejorar el algoritmo evitando comparar innecesariamente aquellos elementos que ya quedaron ordenados.

Es decir, en la primera iteración comparamos todos los elementos del *array* hasta llegar a comparar `arr[len-2]` con `arr[len-1]`. En la segunda iteración comparamos todos los elementos pero solo hasta comparar `arr[len-3]` con `arr[len-2]` porque, como ya sabemos, en `arr[len-1]` se encuentra el elemento que definitivamente ocupará esa posición.

En otras palabras, podemos comparar `arr[i]` con `arr[i+1]` si hacemos variar a `i` entre 0 y `len-j-1` siendo `j` una variable cuyo valor se inicializa en 0 y se incrementa luego de cada iteración.

El desarrollo de esta variante del algoritmo queda a cargo del lector pero a título informativo podemos comentar que una implementación ejecutada sobre la misma computadora mencionada más arriba arrojó los siguientes resultados al ordenar *arrays* de 10, 100, 1000, 10000, 100000 y 1000000 elementos:

```
0 milisegundos (0 minutos, 0 segundos) true
0 milisegundos (0 minutos, 0 segundos) true
0 milisegundos (0 minutos, 0 segundos) true
94 milisegundos (0 minutos, 0 segundos) true
9140 milisegundos (0 minutos, 9 segundos) true
951219 milisegundos (15 minutos, 51 segundos) true
```

El rendimiento mejoró un 300% respecto de la implementación anterior, incluso logró terminar de ordenar el *array* de 1 millón de elementos, aunque demoró algo más de 15 minutos.

Si bien la mejora es importante, el rendimiento del algoritmo continúa siendo inaceptable para ordenar *arrays* de gran tamaño.

20.3 Selection sort (ordenamiento por selección)

El algoritmo de ordenamiento por selección es verdaderamente simple y consiste en recorrer el *array* buscando el menor elemento para intercambiarlo con el primero. Luego recorrer el *array* pero comenzando desde la segunda posición para buscar el menor elemento e intercambiarlo por el segundo y así sucesivamente.

Es decir que si consideramos un *array* `arr` y un índice `i=0`, debemos buscar el menor elemento de `arr` entre las posiciones `i` y `len`-1 e intercambiarlo con `arr[i]`. Luego incrementamos `i` para descartar el primer elemento porque, obviamente, ya contiene su valor definitivo.

En el siguiente gráfico vemos cómo se ordena un *array* `arr = {3, 4, 1, 0, 2}` luego de `len`-1 iteraciones.

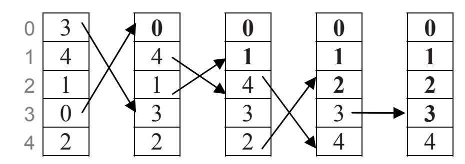


Fig. 20.3 Ordenamiento mediante *selection sort*.

El algoritmo de ordenamiento por selección es de orden cuadrático $O(n^2)$ y tiene un rendimiento similar al de *bubble sort* optimizado.

20.4 Insertion sort (ordenamiento por inserción)

Pensemos en un *array* inicialmente vacío. Luego, cualquier elemento que le agreguemos (llamémosle *e1*) ocupará la primera posición y el *array* estará ordenado. Si agregamos otro elemento (digamos *e2*), este deberá ubicarse antes o después de *e1* según se verifique o no que *e2*<*e1*.

Considerando el *array* `arr = {6, 4, 8, 15, 7, 5, 2}` y un índice `i=1`, entonces llamaremos `s` al *subarray* formado por los todos los elementos de `arr` ubicados entre las posiciones 0 e `i-1`. Es decir, inicialmente será:

`i=1`, entonces:

```
arr = {6, 4, 8, 15, 7, 5, 2}  // arr[i] esta resaltado en negrita
  s = {6}
```

Ahora busquemos el primer elemento de `s` que resulte ser mayor que `arr[i]`. En nuestro ejemplo `s[0]` (que vale 6) es mayor que `arr[i]` (que vale 4); entonces los permutamos:

```
arr = {4, 6, 8, 15, 7, 5, 2}
  s = {4}
```

Ahora incrementamos `i` dejando su valor en 2, entonces:

```
arr = {4, 6, 8, 15, 7, 5, 2}
  s = {4, 6}
```

Como no existe ningún elemento de `s` que resulte ser mayor que `arr[i]` no necesitaremos realizar ninguna permutación; solo incrementaremos `i` dejándolo en 3.

```
arr = {4, 6, 8, 15, 7, 5, 2}
  s = {4, 6, 8}
```

Estamos en la misma situación. No necesitaremos permutar, solo incrementaremos `i` dejándolo en 4.

```
arr = {4, 6, 8, 15, 7, 5, 2}
  s = {4, 6, 8, 15}
```

El primer elemento de `s` que resulta ser mayor que 7 es `s[2]` (cuyo valor es 8). Luego permutamos `s[2]` por `arr[i]`.

```
arr = {4, 6, 7, 15, 8, 5, 2}
  s = {4, 6, 7, 15}
```

Notemos que si ahora incrementamos `i` incluiremos al 8 en el *subarray* `s` y este quedará desordenado. Por lo tanto, en este caso no debemos incrementar el valor de `i`, lo que, en una nueva iteración, nos garantizará permutar el 8 por el 15.

Luego, `i` continúa valiendo 4, entonces:

```
arr = {4, 6, 7, 15, 8, 5, 2}
  s = {4, 6, 7, 15}
```

Ahora el primer valor de `s` que resulta ser mayor que `arr[i]` es 15, ubicado en `s[3]`. Los permutamos:

```
arr = {4, 6, 7, 8, 15, 5, 2}
  s = {4, 6, 7, 8}
```

Luego incrementamos *i* dejándolo en 5:

```
arr = {4, 6, 7, 8, 15, 5, 2}
  s = {4, 6, 7, 8, 15}
```

Y continuamos así sucesivamente hasta que el *subarray* `s`, que no es otra cosa que una vista parcial del *array* `arr`, quede totalmente ordenado.

El algoritmo de ordenamiento por inserción tiene una complejidad cuadrática $O(n^2)$ para el peor de los casos, pero su rendimiento puede mejorar si el *array* que queremos ordenar está parcialmente ordenado.

Más adelante estudiaremos el algoritmo de ordenamiento *Shell* que, valiéndose de una generalización del ordenamiento por inserción, le aporta a este una mejora importante.

20.5 Quicksort (ordenamiento rápido)

Quicksort es un algoritmo relativamente simple y extremadamente eficiente cuya lógica es recursiva y, según su implementación, puede llegar a requerir el uso de *arrays* auxiliares.

20.5.1 Implementación utilizando arrays auxiliares

Llamemos `arr` al *array* que queremos ordenar.

La idea es la siguiente: tomamos cualquier elemento de `arr`. A este elemento lo llamaremos `pivote`. Luego recorremos `arr` para generar dos *arrays* auxiliares: el primero tendrá aquellos elementos de `arr` que resulten ser menores que `pivote`. El segundo tendrá los elementos de `arr` que sean mayores que `pivote`. A estos *arrays* auxiliares los llamaremos, respectivamente, `menores` y `mayores`.

Ahora repetimos el procedimiento, primero sobre `menores` y luego sobre `mayores`.

Finalmente obtenemos el *array* ordenado uniendo `menores + pivote + mayores`.

Por ejemplo: `arr = {4, 7, 2, 5, 1, 9, 3, 8}`

Si consideramos `pivote = 4` (es decir, el primer elemento de `arr`), entonces:

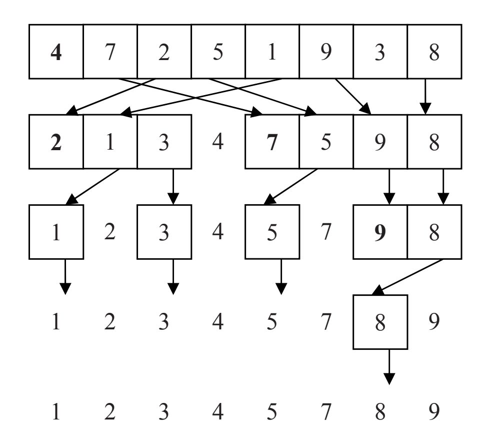


Fig. 20.4 Ordenamiento mediante *quicksort*.

20.5.2 Implementación sin arrays auxiliares

Otra implementación de *quicksort* puede ser la siguiente: luego de seleccionar el elemento `pivote` movemos todos los elementos menores a su izquierda y todos los elementos mayores a su derecha. Esto podemos lograrlo utilizando dos índices: `i`, inicialmente apuntando al primer elemento del *array*, y `j` apuntando al último.

La idea es recorrer el *array* desde la izquierda hasta encontrar el primer elemento mayor que `pivote`. Luego recorrer desde la derecha hasta encontrar el primer elemento menor que el `pivote` y permutarlos.

Por último, invocamos recursivamente dos veces al algoritmo, primero pasándole el *subarray* comprendido entre el inicio y la posición que ocupa el `pivote` (no inclusive) y luego pasándole el *subarray* formado por los elementos que se encuentran ubicados en posiciones posteriores a la del `pivote`. Por ejemplo, si `pivote` es 3, entonces:

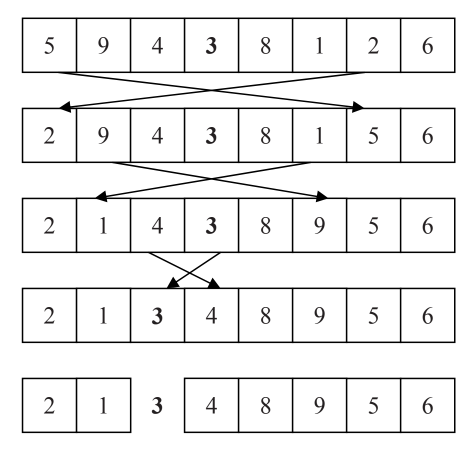


Fig. 20.5 Partición del *array* en elementos menores, pivote y mayores.

Luego de este proceso todos los elementos menores que `pivote` quedarán ubicados a su izquierda mientras que todos los elementos mayores quedarán ubicados a su derecha. Notemos que `pivote` quedó ubicado en su lugar definitivo. El próximo paso será repetir el proceso sobre cada uno de estos *subarrays*.

La complejidad algorítmica de *quicksort* es, en el promedio de los casos: *n log*(*n*).



Algoritmo heapsort, ordenamiento por montículos

20.6 Heapsort (ordenamiento por montículos)

Heapsort es un algoritmo de ordenamiento no recursivo de orden *O(n log n)*) que se basa en la propiedad de "montículo" (*heap* en inglés) de los árboles binarios semicompletos. Por esto, antes de analizar el algoritmo, veremos algunos conceptos sobre árboles binarios que, adrede, no fueron tratados en el capítulo correspondiente.

20.6.1 Árbol binario semicompleto

Decimos que un árbol binario es semicompleto cuando todos sus niveles están completos a excepción del último cuyos nodos deben aparecer de izquierda a derecha.

El siguiente árbol binario es semicompleto.

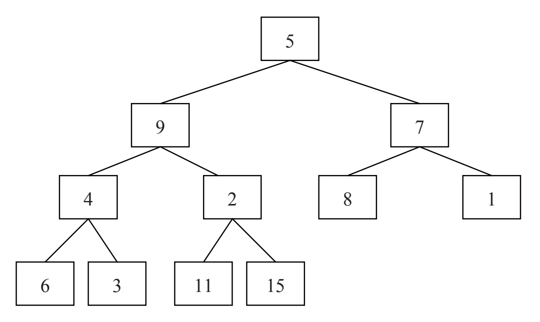
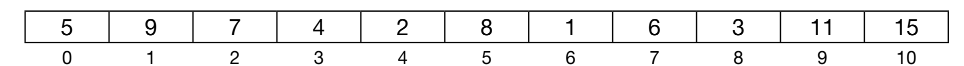


Fig. 20.6 Árbol binario semicompleto.

20.6.2 Representar un árbol binario semicompleto en un array

Un árbol binario semicompleto puede representarse en un *array* asignando en posiciones consecutivas los valores de cada uno de sus nodos, tomándolos de arriba hacia abajo y de izquierda a derecha. Es decir: en la posición 0 del *array* colocamos el valor del nodo raíz, en las posiciones 1 y 2, colocamos a sus hijos izquierdo y derecho y así, sucesivamente.



5	9	7	4	2	8	1	6	3	11	15
0	1	2	3	4	5	6	7	8	9	10

Fig. 20.7 Representación de un árbol binario semicompleto en un *array*.

Calcular qué posiciones ocupan los hijos izquierdo y derecho de un nodo padre

Las posiciones que ocupan, en el *array*, los hijos izquierdo y derecho de cualquier nodo *i* se pueden calcular de la siguiente manera:

izq(i) = 2**i*+1

der(i) = 2**i*+2

Así, en la posición 0 del *array*, encontramos el valor del nodo raíz del árbol (5). Su hijo izquierdo (9) está en la posición: *izq*(0) = 2*0+1 = 1 y su hijo derecho (7) está en la posición: *der*(0) = 2*0+2 = 2. El hijo izquierdo de 9 lo encontraremos en la posición: *izq*(1) = 2*1+1 = 3 y su hijo derecho estará en la posición: *der*(1) = 2*1+2 = 4. Todo esto siempre y cuando 2**i* sea menor o igual que *n*, siendo *n* la posición del último elemento del *array*; que, en este caso es 10.

Calcular la posición del último padre

Volvamos a la Fig. 20.6, que ilustra el árbol binario semicompleto, y observemos que el último nodo padre es 2, ya que este es el que se encuentra más abajo y más a la derecha.

En el *array* el nodo 2 está ubicado en la posición 4, que coincide con *n*/2-1 siendo *n* la posición del último elemento del *array*.

20.6.3 Montículo (*heap*)

Un montículo es un árbol binario semicompleto que tiene la característica de que el valor de cada nodo padre resulta ser mayor que el valor de cualquiera de sus hijos. En este caso, diremos que se trata de un "montículo de máxima". Análogamente, podemos hablar de un "montículo de mínima" cuando el valor de cada nodo padre es menor que los valores de sus hijos. Aquí, solo trabajaremos con montículos de máxima.

Por ejemplo, el siguiente árbol binario es montículo porque, además de ser semicompleto, se verifica que el valor de cada nodo padre es mayor al valor de cada uno de sus hijos.

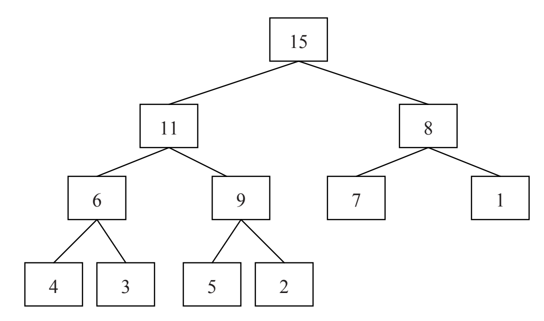


Fig. 20.8 Montículo: árbol binario semicompleto donde cada padre es mayor que sus hijos.

20.6.4 Transformar un árbol binario semicompleto en un montículo

El algoritmo consiste en procesar cada terna {padre, hijo-izquierdo, hijo-derecho} para permutar el valor del padre por el valor del mayor de sus hijos; salvo, que el padre ya sea el mayor de los tres.

El proceso debe ser secuencial, comenzando desde el último padre, luego el anteúltimo y así, sucesivamente, hasta llegar a procesar la raíz.

Volvamos a la Fig. 20.6 que ilustra el árbol binario semicompleto. Vemos que el último padre es 2; luego, al permutarlo por el mayor de sus hijos obtendremos la terna: {15, 11, 2}. El siguiente padre (avanzando hacia la izquierda) es 4 que, al permutarlo por el mayor de sus hijos nos dará la terna: {6, 4, 3}. El próximo padre que debemos considerar es 7 que, luego de procesar su valor y el de sus hijos formará la terna: {8, 7, 1}.

El siguiente padre que corresponde procesar es 9 pero, antes de analizarlo veremos como quedó el árbol binario luego de haber aplicado todas estas permutaciones.

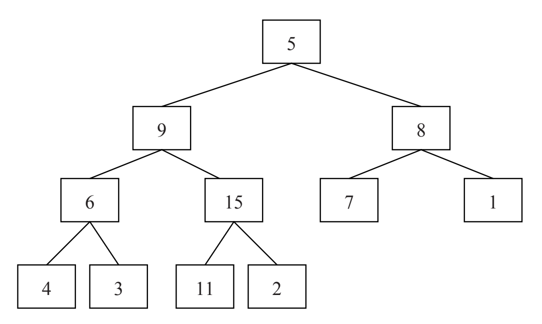


Fig. 20.9 Árbol binario resultante luego de permutar: 2 por 15, 4 por 6 y 7 por 8.

Ahora podemos procesar el nodo 9 y obtener la terna: {15, 6, 9}. El problema es que, luego de esto, 9 quedará posicionado como padre de 11 y 2 haciendo que esta rama del árbol deje de ser montículo.

Para solucionarlo, cada vez que permutemos un padre por alguno de sus hijos, repetiremos el proceso en cascada hacia abajo. Es decir, si permutamos 15 por 9 entonces reconsideraremos la terna que se forma entre el valor permutado (9) y sus (nuevos) hijos. En este caso debemos reconsiderar la terna: {9, 11, 2} y, obviamente, la permutaremos así: {11, 9, 2}. Veamos el estado actual del árbol, donde aún queda pendiente el proceso del nodo raíz.

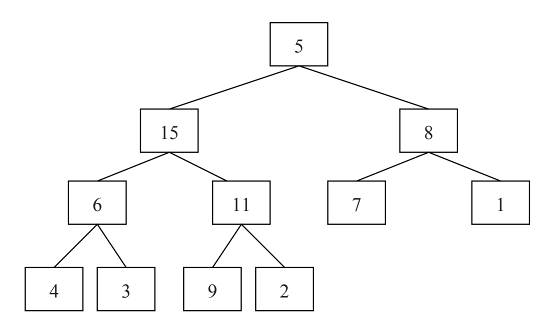


Fig. 20.10 Árbol binario, con todos los nodos procesados excepto la raíz.

Por último, queda por procesar la terna raíz: {5, 15, 8}. Al permutar: 5 por 15, posicionaremos a 5 como padre de 6 y 11. Esto nos obligará a reprocesar el elemento permutado, así que evaluaremos la terna: {5, 6, 11} y permutaremos: 5 por 11. Esta permutación, ahora, ubicó a 5 como padre de 9 y 2, obligándonos a procesar la terna: {5, 9, 2} para finalizar permutando: 5 por 9.

Con esto, el árbol quedó convertido en montículo.

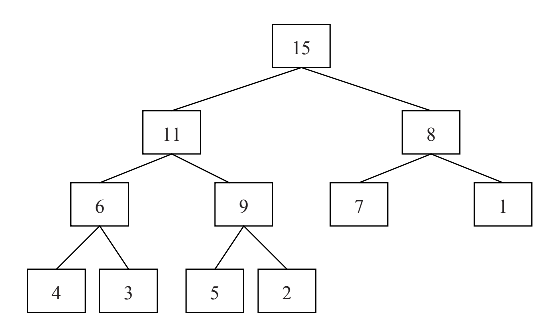


Fig. 20.11 Procesados todos los nodos, el árbol ahora es montículo.

20.6.5 El algoritmo de ordenamiento por montículos

En realidad, el árbol binario no existe como tal. Es solo una manera de imaginar una relación de jerarquía entre los elementos del *array* que queremos ordenar.

Dado que, en un montículo, el mayor valor se ubica en la raíz del árbol; en el *array* dicho valor se ubicará en la posición 0. Entonces, el algoritmo de ordenamiento consiste en los siguientes pasos:

1. Crear el montículo.
2. Colocar al elemento de mayor valor en su posición definitiva.
3. Reconstruir el montículo.

Los pasos 2 y 3 se repiten sucesivamente hasta que el *array* quede ordenado. Notemos que el paso 2 consiste en permutar `arr[0]` por `arr[n]` y luego, decrementar el valor de *n* para dejar de considerar al elemento que ocupa dicha posición.

Aplicaremos este algoritmo para ordenar el siguiente *array*:

```
arr = {5, 9, 7, 4, 2, 8, 1, 6, 3, 11, 15}
```

20.6.5.1 Paso 1 – Crear el montículo

Comenzaremos procesando el último padre que, como dijimos más arriba, se ubica en `arr[`*n*`/2-1]` donde *n* es la posición del último elemento del *array*; en este caso *n*=10.

Procesamos la terna: {2, 11, 15} y permutamos: 2 por 15.

5	9	7	4	**2**	8	1	6	3	**11**	**15**
0	1	2	3	4	5	6	7	8	9	10

5	9	7	4	**15**	8	1	6	3	11	**2**
0	1	2	3	4	5	6	7	8	9	10

Fig. 20.12 Procesamos: {2, 11, 15} y permutamos: 2 por 15.

Ahora procesaremos el anteúltimo padre; la terna: {4, 6, 3} y permutamos: 4 por 6.

5	9	7	**4**	15	8	1	**6**	**3**	11	2
0	1	2	3	4	5	6	7	8	9	10

5	9	7	**6**	15	8	1	**4**	3	11	2
0	1	2	3	4	5	6	7	8	9	10

Fig. 20.13 Procesamos: {4, 6, 3} y permutamos: 4 por 6.

Procesamos la terna: {7, 8, 1} y permutamos: 7 por 8.

5	9	**7**	6	15	**8**	**1**	4	3	11	2
0	1	2	3	4	5	6	7	8	9	10

5	9	**8**	6	15	**7**	1	4	3	11	2
0	1	2	3	4	5	6	7	8	9	10

Fig. 20.14 Procesamos: {7, 8, 1} y permutamos: 7 por 8.

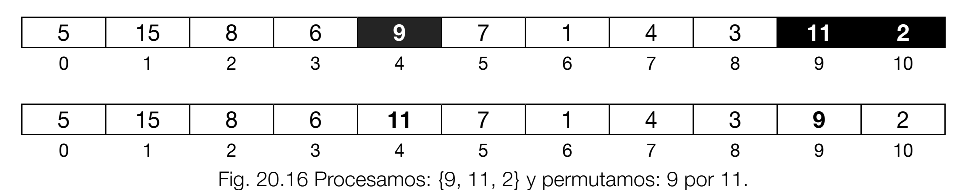
Procesamos la terna: {9, 6, 15} y permutamos: 9 por 15.

5	**9**	8	**6**	**15**	7	1	4	3	11	2
0	1	2	3	4	5	6	7	8	9	10

5	**15**	8	6	**9**	7	1	4	3	11	2
0	1	2	3	4	5	6	7	8	9	10

Fig. 20.15 Procesamos: {9, 6, 15} y permutamos: 9 por 15.

Ahora 9 quedó cómo padre de 11 y 2. Por esto, para que el subárbol continúe siendo montículo debemos procesar también la terna: {9, 11, 2} y permutar: 9 por 11.

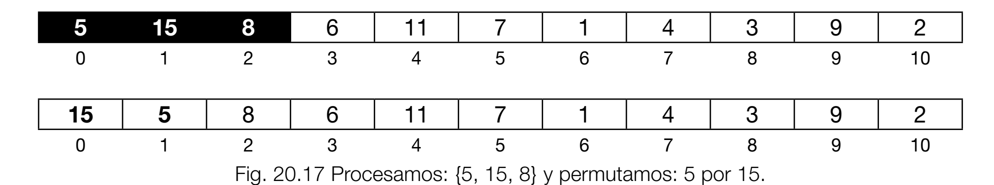


5	15	8	6	**9**	7	1	4	3	**11**	**2**
0	1	2	3	4	5	6	7	8	9	10

5	15	8	6	**11**	7	1	4	3	**9**	2
0	1	2	3	4	5	6	7	8	9	10

Fig. 20.16 Procesamos: {9, 11, 2} y permutamos: 9 por 11.

Finalmente, procesamos la terna: {5, 15, 8} y permutamos: 5 por 15.

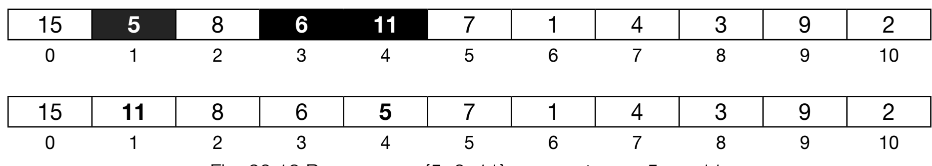


5	**15**	**8**	6	11	7	1	4	3	9	2
0	1	2	3	4	5	6	7	8	9	10

15	**5**	8	6	11	7	1	4	3	9	2
0	1	2	3	4	5	6	7	8	9	10

Fig. 20.17 Procesamos: {5, 15, 8} y permutamos: 5 por 15.

Ahora 5 se posicionó como padre de 6 y 11. Esto nos obliga a procesar la terna: {5, 6, 11} para permutar: 5 por 11.



15	**5**	8	**6**	**11**	7	1	4	3	9	2
0	1	2	3	4	5	6	7	8	9	10

15	**11**	8	6	**5**	7	1	4	3	9	2
0	1	2	3	4	5	6	7	8	9	10

Fig. 20.18 Procesamos {5, 6, 11} y permutamos 5 por 11.

Pero ahora 5 quedó como padre de 9 y 2. Luego, debemos procesar la terna: {5, 9, 2} y permutar: 5 por 9.

15	11	8	6	**5**	7	1	4	3	**9**	**2**
0	1	2	3	4	5	6	7	8	9	10

15	11	8	6	**9**	7	1	4	3	**5**	2
0	1	2	3	4	5	6	7	8	9	10

Fig. 20.19 Procesamos {5, 9, 2} y permutamos 5 por 9.

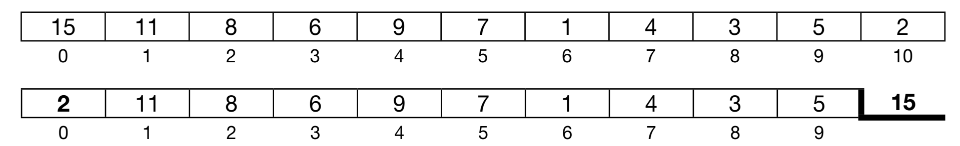
Ahora sí, el *array* representa un montículo ya que el valor de cada nodo padre resulta ser mayor que el valor de sus hijos:

Es decir, se verifica que:

`arr[`*i*`]` > *max*(`arr[`*izq(i)*`]`, `arr[`*der(i)*`]`); para todo *i*<*n*/2;

20.6.5.2 Paso 2 - Colocar al mayor elemento en su posición definitiva

Como en un montículo el elemento de mayor valor se ubica en la raíz, podemos asegurar que en `arr[0]` tenemos el mayor valor del *array*. Luego, al permutar `arr[0]` por `arr[`*n*`]` estaremos ubicando al mayor elemento del *array* en la posición que le corresponde ocupar. Finalmente, debemos decrementar el valor de *n* para que el algoritmo no vuelva a considerar al elemento ubicado en dicha posición.



15	11	8	6	9	7	1	4	3	5	2
0	1	2	3	4	5	6	7	8	9	10

2	11	8	6	9	7	1	4	3	5	**15**
0	1	2	3	4	5	6	7	8	9	

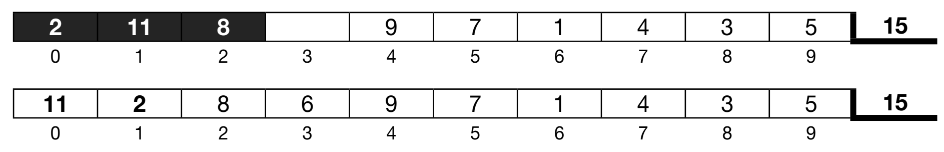
Fig. 20.20 Montículo, donde permutamos `arr[0]` por `arr[n]` y decrementamos *n*.

20.6.5.3 Paso 3 – Reconstruir el montículo

Al permutar `arr[0]` por `arr[n]` el *array* dejó de ser montículo ya que 2 (raíz) no es mayor que sus hijos 11 y 8.

Para reconstruir el montículo aplicaremos el mismo algoritmo con el que lo creamos, solo que ahora no necesitaremos comenzar desde el último padre. Bastará con aplicarlo desde la raíz. Veamos:

Evaluamos la terna: {2, 11, 8} y permutamos: 2 por 11.

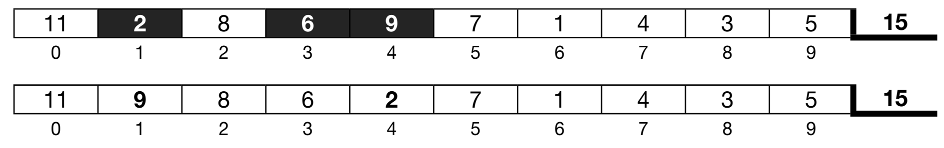


2	**11**	**8**		9	7	1	4	3	5	**15**
0	1	2	3	4	5	6	7	8	9	

11	**2**	8	6	9	7	1	4	3	5	**15**
0	1	2	3	4	5	6	7	8	9	

Fig. 20.21 Procesamos: {2, 11, 8} y permutamos: 2 por 11.

Evaluamos la terna: {2, 6, 9} y permutamos: 2 por 9.



11	**2**	8	**6**	**9**	7	1	4	3	5	**15**
0	1	2	3	4	5	6	7	8	9	

11	**9**	8	6	**2**	7	1	4	3	5	**15**
0	1	2	3	4	5	6	7	8	9	

Fig. 20.22 Procesamos: {2, 6, 9} y permutamos: 2 por 9.

Evaluamos la terna: {2, 5, -} y permutamos: 2 por 5.

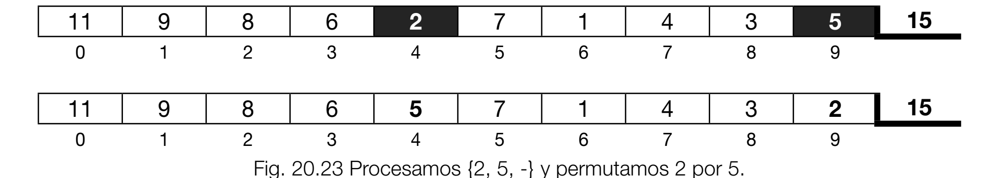


Fig. 20.23 Procesamos {2, 5, -} y permutamos 2 por 5.

Ahora, volviendo al paso 2, al permutar `arr[0]` por `arr[n]` estaremos colocando al (segundo) mayor elemento del *array* en su posición definitiva.

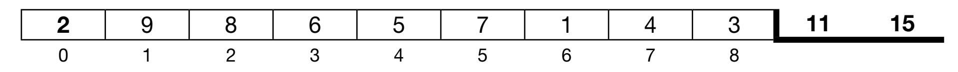


Fig. 20.24 Permutamos: `arr[0]` por `arr[n]` y decrementamos ***n***.

20.7 Shellsort (ordenamiento Shell)

Este método de ordenamiento de complejidad cuadrática para el peor caso surge de una generalización del método de ordenamiento por inserción.

El algoritmo consiste en dividir al *array* en varios *subarrays* más pequeños formados por aquellos elementos del *array* original que se encuentran separados entre sí por una determinada “distancia de paso”.

Por ejemplo, sea el *array* `arr` con los siguientes valores:

```
arr = {23, 45, 4, 17, 7, 3, 15, 24, 12, 21, 9, 1, 6, 11, 5}
```

Entonces, si consideramos una distancia de paso = 7 podemos obtener los siguientes *subarrays*:

```
arr1 = {23, 45,  4, 17,  7, 3, 15}
arr2 = {24, 12, 21,  9,  1, 6, 11}
arr3 = { 5                       }
```

Como vemos, las columnas de esta especie de matriz contienen elementos que, en el *array* original, se encuentran separados por la distancia de paso que definimos más arriba.

Ahora ordenamos las columnas de la matriz:

```
arr1 = { 5, 12,  4,  9,  1, 3, 11}
arr2 = {23, 45, 21,  17, 7, 6, 15}
arr3 = {24                       }
```

Reacomodemos los elementos en el *array* original:

```
arr = {5, 12, 4, 9, 1, 3, 11, 23, 45, 21, 17, 7, 6, 15, 24}
```

Dividamos nuevamente el *array* considerando ahora una distancia de paso = 3.

```
arr1 = { 5, 12,  4}
arr2 = { 9,  1,  3}
arr3 = {11, 23, 45}
arr4 = {21, 17,  7}
arr5 = { 6, 15, 24}
```

Si ordenamos las columnas de la matriz quedará así:

```
arr1 = { 5,  1,  3}
arr2 = { 6, 12,  4}
arr3 = { 9, 15,  7}
arr4 = {11, 17, 24}
arr5 = {21, 23, 45}
```

Unifiquemos una vez más el *array*:

```
arr = { 5, 1, 3, 6,12, 4, 9,15, 7,11,17,24,21,23,45}
```

Comparemos el estado inicial del *array* `arr` con su estado actual.

Original:

```
arr = {23,12, 4, 9, 7, 3,15,24,45,21,17, 1, 6,11, 5}
```

Actual:

```
arr = { 5, 1, 3, 6,12, 4, 9,15, 7,11,17,24,21,23,45}
```

Aunque el *array* aún no está ordenado podemos ver que en la versión actual los valores se encuentran bastante más ordenados que en la versión original.

Si ahora repetimos la operación considerando una distancia de paso = 1, finalmente terminaremos ordenando el *array* por el método de inserción pero aplicado sobre un *array* más ordenado que el original, con lo que el rendimiento del algoritmo será mejor.

Notemos que la distancia de paso que definimos inicialmente (7) coincide con $n/2$ siendo n la longitud del *array*. Luego, la segunda distancia de paso escogida fue 3, que coincide con la mitad de la distancia de paso anterior. Al volver a dividir la distancia de paso tendremos una distancia de paso igual a 1, que representa el caso particular del algoritmo de ordenamiento por inserción.

Este algoritmo no requiere del uso de memoria adicional. Para su implementación podemos pensar en una función `nextElement` que reciba la distancia de paso y la posición actual y retorne la posición del siguiente elemento del *subarray* para esa distancia de paso.

```
public static int next (int distPaso, int posActual)
{
   return posActual+distPaso;
}
```

Análogamente, la función `prevElement` retornará la posición del elemento anterior al especificado.

```
public static int prevElement(int distPaso, int posActual)
{
   return posActual-distPaso;
}
```

20.8 Binsort (ordenamiento por cajas)

Este algoritmo consiste en distribuir los elementos del *array* que queremos ordenar en diferentes “cajas”. Cada caja clasifica los elementos según una determinada propiedad o condición que, obviamente, debe ser mutuamente excluyente para asegurar que cada elemento del *array* ingrese en una única caja.

Por ejemplo, sea el siguiente *array*:

```
arr = {23, 45, 4, 17, 7, 3, 15, 24, 12, 21, 9, 1, 6, 11, 5}
```

entonces podríamos considerar las siguientes cajas:
c0 = { x tal que $0 \leq x < 5$ }
c1 = { x tal que $5 \leq x < 10$ }
c2 = { x tal que $10 \leq x < 15$ }
c3 = { x tal que $15 \leq x < 20$ }
c4 = { x tal que $20 \leq x$ }

Veamos cómo quedaría:
c0 = {4, 3, 1}
c1 = {7, 9, 6, 5}
c2 = {12, 11}
c3 = {17, 15}
c4 = {23, 45, 24, 21}

Luego si ordenamos cada una de estas cajas utilizando cualquier algoritmo de ordenamiento, quedarán así:
c0 = {1, 3, 4}
c1 = {5, 6, 7, 9}
c2 = {11, 12}
c3 = {15, 17}
c4 = {21, 23, 24, 45}

Por último, unimos los elementos de las cajas y obtenemos el *array* ordenado.

20.9 Radix sort (ordenamiento de raíz)

Radix sort es diferente a todos los otros algoritmos ya que su estrategia de ordenamiento es netamente computacional. El algoritmo permite ordenar un conjunto de valores numéricos en función del valor ASCII de cada uno de sus dígitos.

Pensemos en el siguiente *array*:

```
arr = {12, 5, 136, 432, 226, 125, 62, 461, 25, 91}
```



Nota: Tenga en cuenta, el lector, que tanto en C como en Java cuando se antepone el prefijo 0 (cero) a un valor entero, este será interpretado como un número octal. Por esta razón, los ejemplos aquí expuestos no deben ser considerados de manera literal, ya que la implementación de este algoritmo requerirá que cada número entero que se procese, primero sea convertido a cadena de caracteres

Como vamos a procesar "dígito a dígito" los elementos del *array*, todos ellos deben estar expresados con la misma cantidad de dígitos.

```
arr = {012, 005, 136, 432, 226, 125, 062, 461, 025, 091}
```

Ahora todos los elementos de `arr` tienen la misma cantidad de dígitos.

Comenzaremos por distribuir los números del *array* en diferentes listas que permitan clasificarlos según su último dígito.

```
L(0) = {}               // ningun elemento termina en 0
L(1) = {461, 091}       // estos terminan en 1
L(2) = {012, 432, 062}  // estos terminan en 2
L(3) = {}               // ninguno termina en 3
L(4) = {}               // ninguno termina en 4
```

```
L(5) = {005, 125, 025}  // estos terminan en 5
L(6) = {136, 226}       // :
L(7) = {}
L(8) = {}
L(9) = {}
```

Rearmemos el *array*:

```
arr = {461, 091, 012, 432, 062, 005, 125, 025, 136, 226}
```

Repetimos la operación considerando el segundo dígito menos significativo:

```
L(0) = {005} // este es el unico que tiene 0 en la anteultima posicion
L(1) = {012} // este es el unico que tiene 1 en la anteultima posicion
L(2) = {125, 025, 226} // estos tienen 2 en la anteultima posicion
L(3) = {432, 136}      // estos tienen 3 en la anteultima posicion
L(4) = {}              // ninguno tiene 4 en la anteultima posicion
L(5) = {}              // :
L(6) = {461, 062}
L(7) = {}
L(8) = {}
L(9) = {091}
```

Rearmemos el *array*:

```
arr = {005, 012, 125, 025, 226, 432, 136, 461, 062, 091}
```

Repetimos la operación con el tercer dígito menos significativo.

```
L(0) = {005, 012, 025, 062, 091}
L(1) = {125, 136}
L(2) = {226}
L(3) = {}
L(4) = {432, 461}
L(5) = {}
L(6) = {}
L(7) = {}
L(8) = {}
L(9) = {}
```

Ahora al rearmar el *array* los números se encuentran ordenados.

```
arr = {005, 012, 025, 062, 091, 125, 136, 226, 432, 461}
```

Radix sort es un algoritmo altamente eficiente y su complejidad es lineal $O(n)$.

20.9.1 Ordenar cadenas de caracteres con radix sort

Radix sort permite ordenar cadenas de caracteres ya que cada carácter tiene asignado un valor numérico definido en la tabla ASCII. En este caso debemos completar las cadenas más cortas con espacios en blanco a la derecha para que todas tengan la misma longitud.

Si vamos a comparar las cadenas "Juan" con "Alberto" debemos tener en cuenta lo siguiente:

Correcto	**Correcto**	~~**Incorrecto**~~
[Juan]	[Juan]	[Juan]
[Alberto]	[Alberto]	[Alberto]

20.10 Resumen

En este capítulo analizamos diferentes métodos de ordenamiento clasificándolos según su complejidad y llegamos a la conclusión de que los algoritmos de orden cuadrático no funcionan bien cuando la cantidad de elementos que se va a ordenar es elevada.

En general, se considera que el algoritmo de ordenamiento "por excelencia" es *quicksort* por ser relativamente fácil de implementar y extremadamente eficiente. De hecho, Java provee una implementación de este algoritmo dentro del método estático `Collections.sort`.

En el próximo capítulo analizaremos las características de algunos de los algoritmos estudiados a lo largo de este libro para encontrar patrones comunes que nos permitan clasificarlos según un criterio que llamaremos estrategia algorítmica.

20.11 Contenido de la página Web de apoyo



El material marcado con asterisco (*) solo está disponible para docentes.

20.11.1 Mapa conceptual

20.11.2 Autoevaluaciones

20.11.3 Videotutorial

20.11.3.1 Algoritmo heapsort, ordenamiento por montículos

20.11.4 Presentaciones*

21

Estrategia algorítmica

Contenido

Objetivos del capítulo

- Descubrir características comunes que a los diferentes algoritmos, que nos permitan reconocer la existencia de patrones para clasificarlos según su estrategia de resolución.
- Conocer los principales modelos de estrategia algorítmica: "divide y conquista", "*greddy*" y "programación dinámica".

Competencias específicas

- Identificar las características comunes de diferentes algoritmos, para reconocer patrones y facilitar la creación de programas.



En la Web de apoyo del libro encontrará el contenido de este capítulo para leer online:

1. Ir a la pagina http://virtual.alfaomega.com.mx
2. Ingresar los nombres de Usuario y Contraseña definidos cuando registro el libro (ver Registro en la Web de apoyo).
3. Hacer un clic en el hipervínculo que lleva el nombre del capítulo que desea leer.

22

Algoritmos sobre grafos

Contenido

Objetivos del capítulo

- los principales problemas que pueden ser representados mediante el uso de grafos y estudiar los algoritmos que los resuelven: Dijkstra, Prim, Kruskal.
- Comparar las implementaciones “*greddy*” y “dinámica” de algunos de estos algoritmos.

Competencias específicas

- Analizar y resolver problemas de la vida real que pueden ser representados con grafos.



En la Web de apoyo del libro encontrará el contenido de este capítulo para leer online:

1. Ir a la pagina http://virtual.alfaomega.com.mx
2. Ingresar los nombres de Usuario y Contraseña definidos cuando registro el libro (ver Registro en la Web de apoyo).
3. Hacer un clic en el hipervínculo que lleva el nombre del capítulo que desea leer.

Bibliografía

- Eckel, Bruce - *Piensa en Java*, Prentice Hall, España, 4a ed., 2007.
- Joyanes Aguilar, Luis y Zahonero Martínez, Ignacio - *Programación en C, Metodología, Algoritmos y Estructura de Datos,* McGraw Hill, España, 2a ed., 2005.
- Kernighan, Brian W. y Ritchie, Dennis M. - *El Lenguaje de Programación C,* Prentice Hall, México, 2a ed., 1991.
- López, Gustavo; Jeder, Ismael y Vega, Augusto - *Análisis y Diseño de Algoritmos, Implementaciones en C y Pascal*, Argentina, 2009.
- Stroustrup, Bjarne - *El Lenguaje de Programación C++,* Addison-Wesley, España, 2002.
- Sznajdleder, Pablo - *HolaMundo.Pascal, Algoritmos y Estructuras de Datos,* Editorial del CEIT, Argentina, 2008.
- Sznajdleder, Pablo - *Java a Fondo, Estudio del Lenguaje y Desarrollo de Aplicaciones,* Alfaomega Grupo Editor, Argentina, 2010.